System Architecture
as an Ordinary Engineering Discipline
Version 0.58

Christoph Baumann & Wolfgang J. Paul & Sabine Schmaltz

July 24, 2013

In case of discovery of errors or difficulty understanding specific parts of the book, please report to sabine@wjpserver.cs.uni-saarland.de. Our goal is to bring this book in the best possible state we can – before it is published. Thus, any input from students is both important and helpful.

Bitte bei Fehlern oder unverständlichen Textpassagen einfach per eMail (oder persönlich vor oder nach der Vorlesung) bei sabine@wjpserver.cs.uni-saarland.de melden. Ziel ist es das Buch vor Veröffentlichung in den bestmöglichen Zustand zu bringen, daher ist jeder Input von Studenten wichtig und nützlich.

(v0.16) Clarified chapter 1 by using the successor-function S (instead of $+1$) where appropriate.

(v0.17) Corrected the definition of $nzero$.

(v0.18) Changed the definition of u on page 103 to match the lecture. Corrected BCE signals according to the lecture.

(v0.19) Corrected the text on page 31, "half adder"→"full adder"

(v0.20) Corrected definitions of $alur$ and $alui$ (page 121).

(v0.21) Renamed MIPS configurations c to configurations d.

Corrected the definition of jump targets.

Corrected several minor typos.

Renamed the access width d to a (because $d(d)$ looks confusing).

(v0.22) Undid the renaming since the same MIPS chapter is also used in the MultiCore-book.

(v0.23) Corrected $rs(c)$ to $rd(c)$ on page 126. $gpr(c)$ to $gpr(x)$ on page 130. Typo page 132 $foralla \rightarrow \forall a$

(v0.24) Corrected the picture of the BCE-unit (1-bit output, n- instead of k-bit b input) on page 111.

(v0.25) Corrected several typos in the ALU chapter.

(v0.26) Corrected neg to ovf on page 104.

(v0.27) Corrected page 102 bottom B^n to B_n. Corrected several typos in the first three chapters.

(v0.28) Added chapter 7. Beware of typos, I have not read it yet.

(v0.29) Chapter 7 was extended.

(v0.30) Chapter 7 improved.

(v0.31) A very preliminary chapter 8 was added.

(v0.32) Added missing null-pointer-constant.

(v0.33) Chapter 8 extended.

(v0.34) Further extensions to chapter 8.

(v0.35) Boolean expressions added as parameters to functions.

(v0.36) C0 chapter extended by expression evaluation.

(v0.37) C0 chapter parts of statement execution added, fixed a few language errors.

(v0.38) Further extensions to chapter 8.

(v0.39) Lots of typos corrected.

(v0.40) Several minor clarifications.

(v0.41) Improvements to the text/style of presentation of Chapter 1. Errors fixed in several chapters.

(v0.42) Editing of Chapter 2, 3, 4, 5. Small errors in several chapters fixed.

(v0.43) Fixed typo in chapter 1 ($a_1 = 9 \rightarrow a_1 = 1$)

(v0.44) Added definition of adding sequence elements left/right (page 20)

(v0.45) Added definition concatenation of sequences

(v0.46) Corrected many small errors in chapter 8, keep in mind, though, that this chapter will be changed significantly during the lecture. Added a small remark on the order of applying lemmas and theorems in a mathematical theory in chapter 1. Due to the switch to Springer monograph document format, there might be several style issues that will need to be resolved.

(v0.47) Added a small paragraph about sequence element numbering convention on page 12. I am not 100% sure this convention is obeyed everywhere. Please report any violation of this condition if you encounter one.

(v0.48) Added a paragraph about the conditional-sum-adder and its construction in the ALU chapter.

(v0.49) Missing $af[3]$ condition added in ALU. Parallel-prefix in basic circuits.

(v0.50) Corrected $\wedge$-gate in parallel prefix to $\circ$-gate.

(v0.51) Corrected *ifill* definition in ISA. Removed irrelevant RAM designs. Reworking MIPS hardware implementation to the simple 2-phase execute-fetch machine of the lecture (this means many things get simpler) – this is only partially done so far! Going back to a layout with bigger font and less margins for the lecture notes (cheaper for printing and carrying around).

(v0.52) Finished adapting construction and correctness proof of the 2-phase machine (had to leave the proof of hardware memory correctness as an exercise due to lack of time to write down real proof). It might be a good idea to get rid of byte- and halfword writes in the ISA specification machine when we don't even implement them.

(v0.53) Rewrite of beginning of C0 chapter. Added missing space character in C0 grammar. Corrected PaDF→PaDS in C0 grammar.

(v0.54) Removed MIPS shift instructions (which we do not implement in this book) from the ISA tables.

(v0.55) Corrected order of typedef on page 139. Removed hardware implementation of shift from the book.

(v0.56) Parallel prefix circuit moved to ALU chapter. Significant improvements in MIPS hardware construction chapter. Rewrite of parts of C0 chapter.

(v0.57) C0 examples partially rewritten. Minor changes elsewhere.

(v0.58) Minor errors corrected. Extended programming examples.

Contents

Chapter 1

Introduction

Dear Reader,

this booklet contains the lecture notes of a class we teach in Saarbrucken to first year students within a single semester. The purpose of the class is simple: to exhibit constructions of

- a simple MIPS processor
- a simple compiler for a C dialect
- a small operating system kernel

and to give detailed explanations why they work.

Clearly, this is much more material on much fewer pages than you would expect to find in a usual textbook. So, where is the secret to so much speed resp. productivity and where is the catch? The secret - if you want to call it this way - is, that we treat computer science as an engineering discipline like any other: to any topic there is appropriate math which is intuitive and completely adequate to deal with it; both with specifications and with explanations why constructions work. Highschool mathematics happens to suffice to deal with all subjects in this booklet. The only catch is that you might have to give up a belief: that this cannot be done. But engineering is about facts, not beliefs.

But then, if speed is what you cherish and if kernels are what you are really interested in: wouldn't it be even faster by skipping processors and compilers and by jumping directly to the third subject? This is the approach taken in traditional texts about operating systems and: no, this is much slower. Kernel construction has foundations, which happen to be processors and compilers. Read the introductions of the corresponding chapters in this booklet for explanation. Building on shaky foundations is slow and the result needs continuous repairs[1].

Are we hinting here that it is possible to construct entire systems that never need repair and which cannot crash except by physical failure of the hardware? Say by

- proving that they always work
- making the proof machine-readable and
- checking with a computer program, that the proof has no gaps.

The simple answer is 'yes'. It is possible and it has been done. For very simple systems as early as xxxx [CLI], for complex processors in 2003 [we], for non optimizing compilers in xxxx[] and for optimizing compilers in xxxx [leroy], for a complete kernel written in C including all lines of assembly code in xxxx[], for the C portions of a fairly complex kernel in 2010 [Nicta]. For the underlying mathematical theory see ... believe it or not ... this booklet. It is exactly the same theory which gives the fastest explanation in the classroom, why systems work.

[1] repairs are called 'updates' for software products

If this is real, you might say, and if this technology scales to the complexity of industrial products, then a certain company with headquarters in Redmond should be interested in this. It certainly is since 2007 [Verisoft-XT].

About stars: this booklet is thin, yet it contains more than the absolutely necessary material. Chapters and sections marked with stars are complementary reading. Even if you skip them you gain a very solid understanding of the entire subject.

For the chapters

- a main purpose of a kernel is to implement multiprocessing. This amounts to simulating on a single processor multiple processors with the same instruction set. If you cannot specify an instruction set and a simulation, you cannot even state precisely, what a kernel is supposed to do.

- kernels perform so called process switches. They save processor registers into and restore processor registers from certain C variables, which are called process control blocks. One cannot program this in C, because processor registers are simply not visible in higher programming languages. One must program this in assembly language, but these assembly portions of the program access C variables (the process control blocks). One cannot even specify the effect of these instructions without referring to the allocation function (address of) of the C compiler used.

Therefore, if you do not have a precise understanding of processors and compilers, then kernel construction is like building the third story of a house on top of a very shaky second floor. You end up repairing and fixing the third story all your life. If your kernel were a product, you might end up sending out updates constantly.

Chapter 2

Understanding Decimal Addition

2.1 Experience versus Understanding

This booklet is about understanding system architecture in a quick and clean way: no black art, nothing you can only get a feeling for after years of programming experience. While experience is good, it does not replace understanding. For illustration, consider the basic method for decimal addition of one digit numbers as taught in the first weeks of elementary school: Everybody has experience with it; *very* few of the people who do not understand it realize that they don't.

Recall that, in mathematics, there are definitions and statements. Statements that we can prove are called theorems. Some true statements, however, are so basic that there are no even more basic statements that we can derive them from; these are called axioms. A person that understands decimal addition will clearly be able to answer the following simple

Questions: Which of the following equations are definitions? Which ones are theorems? If an equation is a theorem, what is the proof?

$$\begin{aligned} 2+1 &= 3 \\ 1+2 &= 3 \\ 9+1 &= 10 \end{aligned}$$

We just stated that these questions are simple; we did not say that answering them is easy. Should you care? Indeed you should, for at least three reasons: i) In case you don't even understand the school method for decimal addition, how can you hope to understand computer systems? ii) The reason, why the school method works has *very* much to do with the reason, why binary addition in the fixed point adders of processors works. iii) You should learn to distinguish between having experience with something that has not gone wrong (yet) and having an explanation of why it always works. The authors of this booklet consider iii) the most important.

2.2 The Natural Numbers

In order to answer the above questions, we first consider counting. Since we count by repeatedly adding 1, this should be a step in the right direction.

The set of *natural numbers* $\mathbb{N}$[1] and the properties of counting are not based on ordinary mathematical definitions. In fact, they are so basic that we use 5 axioms due to Peano to simultaneously lay down all properties about the natural numbers and of counting we will ever use without proof. The axioms talk about

[1]The natural numbers are sometimes also referred to by the term *counting numbers* or, in our case, more precisely, the term *non-negative integers*.

- a special number 0,
- the set $\mathbb{N}$ of all natural numbers (with zero),
- counting formalized by a *successor* function $S : \mathbb{N} \to \mathbb{N}$, and
- subsets $A \subset \mathbb{N}$ of the natural numbers.

Peano's axioms are

1. $0 \in \mathbb{N}$. Zero is a natural number.[2]
2. $x \in \mathbb{N} \to S(x) \in \mathbb{N}$. You can always count to the next number.
3. $x \neq y \to S(x) \neq S(y)$. Different numbers have different successors.
4. $\nexists y : 0 = S(y)$. By counting you cannot arrive at 0. Note, that this isn't true for computer arithmetic, where you **can arrive** at zero by an overflow of modulo arithmetic (see Section 3.2).[3]
5. $A \subset \mathbb{N} \wedge 0 \in A \wedge (n \in A \to S(n) \in A) \to A = \mathbb{N}$. This is the famous induction scheme for proofs by induction. We give plenty of examples later.

In an induction proof, one usually considers a set A consisting of all numbers n satisfying a certain property $P(n)$:

$$A = \{n \in \mathbb{N} \mid P(n)\}$$

Then,

$$\begin{aligned} n \in A &\leftrightarrow P(n) \\ A = \mathbb{N} &\leftrightarrow \forall n \in \mathbb{N} : P(n) \end{aligned}$$

and the induction axiom translates into a proof scheme you might or might not know from high school:

- Start of the induction: show P(0).
- Induction step: show that $P(n)$ implies $P(S(n))$.
- Conclude $\forall n \in \mathbb{N} : P(n)$. Property P holds for all natural numbers.

With the rules of counting laid down by the Peano axioms, we are able to make two 'ordinary' definitions. We define 1 to be the next number after 0 if you count. We also define that addition of 1 is counting.

Definition 1 (Adding 1 by Counting)**.**

$$\begin{aligned} 1 &= S(0) \\ x + 1 &= S(x) \end{aligned}$$

With this, the induction step of proofs by induction can be reformulated to the more familiar form

- Induction step: show that $P(n)$ implies $P(n + 1)$.

[2]This is a modern view of counting, because zero counts something that could be there but isn't.

[3]Should you, dear reader, ever encounter in your future life articles or textbooks on programming or even program correctness modeling the unsigned integers of a programming language by the natural numbers $\mathbb{N}$, then clearly somebody does not even understand the difference between ordinary counting and computer arithmetic. It isn't you.

2.2.1 $2 + 1 = 3$ is a Definition

One can now give meaning to the other digits of decimal numbers with the following mathematical definition.

Definition 2 (The Digits 2 to 9)**.**

$$\begin{aligned} 2 &= 1 + 1 = S(1) \\ 3 &= 2 + 1 = S(2) \\ 4 &= 3 + 1 = S(3) \\ &\vdots \\ 9 &= 8 + 1 = S(8) \end{aligned}$$

Thus, $2 + 1 = 3$ is the definition of 3. In contrast,

2.2.2 $1 + 2 = 3$ is a Theorem

Expanding definitions, we would like to prove it by

$$\begin{aligned} 1 + 2 &= 1 + (1 + 1) \quad \text{(Definition of 2)} \\ &= (1 + 1) + 1 \\ &= 2 + 1 \quad \text{(Definition of 2)} \\ &= 3 \quad \text{(Definition of 3)} \end{aligned}$$

With the axioms and definitions we have so far, we cannot prove the second equation, yet. This is due to the fact that we have not defined addition completely. This is fixed by the following inductive definition:

Definition 3 (Addition)**.**

$$\begin{aligned} x + 0 &= x \\ x + S(y) &= S(x + y) \end{aligned}$$

In words: adding 0 does nothing. In order to add $y + 1$, first add y, then add 1 (by counting to the next number). From this we can derive the usual laws of addition.

Lemma 1 (Associativity of Addition)**.**

$$(x + y) + z = x + (y + z)$$

by induction on z. For $z = 0$ we have

$$(x + y) + 0 = x + y = x + (y + 0)$$

by definition of addition (adding 0).

For the induction step we assume the induction hypothesis $x+(y+z) = (x+y)+z$. By repeatedly applying the definition of addition we conclude

$$\begin{aligned} (x + y) + S(z) &= S((x + y) + z) \quad \text{(by definition of addition)} \\ &= S(x + (y + z)) \quad \text{(by induction hypothesis)} \\ &= x + S(y + z) \quad \text{(by definition of addition)} \\ &= x + (y + S(z)) \end{aligned}$$

□

Substituting $x = y = z = 1$ in Lemma 1, we get

$$(1 + 1) + 1 = 1 + (1 + 1)$$

which completes the missing step in the proof of $1 + 2 = 3$.

Showing the commutativity of addition is surprisingly tricky. We first have to show two special cases.

Lemma 2. $0 + x = x$

by induction on x. For $x = 0$ we have

$$0 + 0 = 0$$

by the definition of addition.

For the induction step we can assume the induction hypothesis $0 + x = x$ and use this to show

$$\begin{aligned} 0 + S(x) &= S(0 + x) \quad \text{(definition of addition)} \\ &= S(x) \quad \text{(induction hypothesis)} \end{aligned}$$

□

Lemma 3. $x + 1 = 1 + x$

by induction on x. For $x = 0$ we have by the previous lemma and the definition of addition

$$0 + 1 = 1 = 1 + 0$$

For the induction step we can assume the induction hypothesis $x + 1 = 1 + x$ and show

$$\begin{aligned} 1 + S(x) &= S(1 + x) \quad \text{(definition of addition)} \\ &= S(x + 1) \quad \text{(induction hypothesis)} \\ &= S(x) + 1 \quad \text{(definition of counting by adding 1)} \end{aligned}$$

□

Lemma 4 (Commutativity of Addition)**.**

$$x + y = y + x$$

by induction on y. For $y = 0$ we have

$$\begin{aligned} x + 0 &= x \quad \text{(definition of addition)} \\ &= 0 + x \quad \text{(lemma 2)} \end{aligned}$$

For the induction step we can assume the induction hypothesis $x + y = y + x$ and show

$$\begin{aligned} x + S(y) &= S(x + y) \quad \text{(definition of addition)} \\ &= S(y + x) \quad \text{(induction hypothesis)} \\ &= y + S(x) \quad \text{(definition of addition)} \\ &= y + (x + 1) \quad \text{(definition of counting by adding 1)} \\ &= y + (1 + x) \quad \text{(lemma 3)} \\ &= (y + 1) + x \quad \text{(associativity of addition)} \\ &= S(y) + x \quad \text{(definition of counting by adding 1)} \end{aligned}$$

□

2.2.3 $9 + 1 = 10$ is a Brilliant Theorem

The proof of $9 + 1 = 10$ is much more involved. It uses a special biological constant defined as

$$Z = 9 + 1$$

which denotes the number of our fingers or, respectively, toes[4]. Moreover it uses a definition attributed to the brilliant Arab mathematician al-Khwarizmi which defines the decimal number system.

Definition 4 (Decimal Numbers)**.** *An n digit decimal number $a_{n-1} \dots a_0$ with digits $a_i \in \{0, \dots, 9\}$ is interpreted as*

$$a_{n-1} \dots a_0 = \sum_{i=0}^{n-1} a_i \cdot Z^i$$

Substituting $n = 2$, $a_1 = 1$ and $a_0 = 0$ we can derive the proof of $9 + 1 = 10$. We must however evaluate the formula obtained from the definition of decimal numbers; in doing so, we need properties of exponentiation and multiplication:

$$\begin{aligned} 10 &= 1 \cdot Z^1 + 0 \cdot Z^0 \quad \text{(definition of decimal number)} \\ &= 1 \cdot Z + 0 \cdot 1 \quad \text{(properties of exponentiation)} \\ &= Z + 0 \quad \text{(properties of multiplication)} \\ &= Z \quad \text{(definition of addition)} \\ &= 9 + 1 \quad \text{(definition of } Z\text{)} \end{aligned}$$

The interested reader will observe that, in elementary school, they were taught only the multiplication of decimal numbers, which relied on decimal addition, and that exponentiation came after multiplication. In this way we cannot possibly fill the gaps in the proof above. Instead, one defines multiplication and exponentiation without relying on decimal numbers, as below.

Definition 5 (Multiplication)**.**

$$\begin{aligned} x \cdot 0 &= 0 \\ x \cdot S(y) &= x \cdot y + x \end{aligned}$$

Definition 6 (Exponentiation)**.**

$$\begin{aligned} x^0 &= 1 \\ x^{S(y)} &= x^y \cdot x \end{aligned}$$

By Lemma 2, we get

$$x \cdot 1 = x \cdot (S(0)) = x \cdot 0 + x = 0 + x = x$$

i.e., multiplication of x by 1 from the right results in x. In order to progress in the proof of $9+1 = 10$, we show

Lemma 5. $1 \cdot x = x$

by induction on x. For $x = 0$ we have $1 \cdot 0 = 0$ by the definition of multiplication. For the induction step we can assume the induction hypothesis $1 \cdot x = x$ and show

$$\begin{aligned} 1 \cdot S(x) &= 1 \cdot x + 1 \quad \text{(definition of multiplication)} \\ &= x + 1 \quad \text{(induction hypothesis)} \end{aligned}$$

□

[4]We use the letter Z here because in German, our native language, the word for 'ten' and for 'toes' is almost the same: 'Zehn' vs. 'Zehen'.

Note that, in all of the lemmas we just proved, one has to be careful to not simply apply basic arithmetic rules known from high school.[5] In order to obtain a sound mathematical theory, a proof of a new lemma can only only use axioms, definitions and lemmas that have been proven before. Indeed, one of the main challenges in establishing a mathematical theory lies in identifying an order of lemmas that allows for the shortest, and thus, considered most beautiful, proofs.

Hints for the proofs of the following laws can be be found in the exercise section.

$$\begin{aligned}(x \cdot y) \cdot z &= x \cdot (y \cdot z) \quad \text{(associativity of multiplication)} \\ x \cdot y &= y \cdot x \quad \text{(commutativity of multiplication)} \\ (x + y) \cdot z &= x \cdot z + y \cdot z \quad \text{(distributivity)}\end{aligned}$$

Using Lemma 5, we get

$$x^1 = x^{S(0)} = x^0 \cdot x = 1 \cdot x = x \tag{2.1}$$

We finish the section by showing a classical identity for exponentiation

Lemma 6. $x^{y+z} = x^y \cdot x^z$

by induction on z. For $z = 0$ we have (leaving the justification of the steps as an exercise)

$$x^{y+0} = x^y = x^y \cdot 1 = x^y \cdot x^0$$

For the induction step, we assume the induction hypothesis $x^{y+z} = x^y \cdot x^z$ and show

$$\begin{aligned}x^{y+S(z)} &= x^{S(y+z)} \quad \text{(definition of addition)} \\ &= x^{(y+z)} \cdot x \quad \text{(definition of exponentiation)} \\ &= (x^y \cdot x^z) \cdot x \quad \text{(induction hypothesis)} \\ &= x^y \cdot (x^z \cdot x) \quad \text{(associativity of multiplication)} \\ &= x^y \cdot (x^{S(z)}) \quad \text{(definition of exponentiation)}\end{aligned}$$

□

2.3 Summary

At the example of decimal addition, we have demonstrated that being used to something that has not gone wrong yet and understanding it are very different things. We have reviewed Peano's axioms and have warned the reader that computer arithmetic does not satisfy them: unsigned integers in computers are not the natural numbers. We have stated the definition of decimal numbers; this will turn out to be helpful when we study binary arithmetic and construct adders in the next chapter. We have practiced proofs by induction and derived the ring axioms for the natural numbers with addition and multiplication as well as a classical identity about exponentiation without referring to the usual decimal number representations.

[5] In the proof of Lemma it might appear valid to just apply the definition "$S(x) = x + 1$", resulting in "$1 \cdot S(x) = 1 \cdot (x + 1) = 1 \cdot x + 1 \cdot 1$. Applying distributivity of natural numbers here, however, would be problematic since it hasn't been proven yet.

Chapter 3

Basic Mathematical Concepts

We begin in Section 3.1 with very basic definitions like intervals $[i : j]$ of natural numbers. So we replace them by fairly obvious inductive definitions. We deal with the minor technical nuisance, that usually sequence elements are numbered from left to right starting with index 1, but in number representations it is much nicer to number them from right to left starting with index 0. We also introduce vector operations on bit strings.

Section 3.2 on modulo arithmetic was included for several reasons. i) The notation $mod \quad k$ is overloaded: it is used to denote both the congruence *relation* modulo a number k or the *operation* of taking remainder of integer division by k. We prefer our readers to clearly understand this. ii) Fixed point arithmetic is modulo arithmetic, so we will clearly have to make use of it. The most important reason however is iii) addition/subtraction of binary numbers and of two's complement numbers is done by exactly the same hardware[1]. When we get to this topic this will look completely intuitive, and therefore there should be a very simple proof justifying this fact. Such a proof will be exhibited later; it hinges on the simple lemma 11, which deals with the solution of congruence equations from this section.

The very short Section 3.3 on geometric and arithmetic sums is simply there to remind the reader of the proof of the classical formulas for the computation of geometric and arithmetic sums, which are much easier to memorize than the formula itself. The formula for geometric sums is used to bound the range of numbers represented by bit strings.

Section 3.4 contains some very basic graph theory. Lemma 13 contains the result, that path length in directed acyclic graphs is bounded. This little result is remarkably important for several reasons: i) its short proof is a great illustration of the pigeon hole principle. ii) it justifiers to define depth of nodes in directed acyclic graphs. This is later used to show - by an induction on the depth of nodes - that the behavior of switching circuits is well defined. iii) it is used to show the equivalence of a recursive definition and the classical graph theoretic definition of rooted trees. This result is later used to relate our formalization of derivation trees for context free grammars with classical graph theory.

The chapter is in a sense like a lexicon with fairly simple reference material. As a reader of the book glimpse over it, notice what is there, but worry about detailed understanding when the material is used. In lectures, just sketch the material of this chapter. Provide details or refer students to the text when the material is used.

[1] Except for the computation of overflow and negative signals.

3.1 Basics

3.1.1 Numbers and Sets

We denote by

$$\mathbb{N} = \{0, 1, 2, \ldots\}$$

the set of natural numbers including zero, by

$$\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$$

the set of integers. Unless explicitly stated otherwise we identify natural numbers with their decimal representation, as we always have done since elementary school.

We denote by

$$\mathbb{B} = \{0, 1\}$$

the set of Boolean values. We also use the term *bit* to refer to a Boolean value.

For integers i, j with $i < j$ we define the interval of integers from i to j by

$$[i : j] = \{i, i + 1, \ldots, j\}$$

Strictly speaking, definitions using three dots are never precise; they resemble intelligence tests, where the author hopes that all readers who are forced to take the test arrive at the same solution. Usually, one can easily find a corresponding and completely precise recursive definition (without three dots) in such a situation. Thus, we define $[i : j]$ in a rigorous way as

$$\begin{aligned} [i : i] &= \{i\} \\ [i : j + 1] &= [i : j] \cup \{j + 1\}. \end{aligned}$$

The Hilbert $\in$-Operator $\in A$ picks an element from a set A. Applied to a singleton set, it returns the unique element of the set:

$$\in\{x\} = x.$$

For finite sets A, we denote by $\#A$ the *cardinality*, i.e. the number of elements in A.

Given a function f operating on a set A and a set $A_1 \subseteq A$, we denote by $f(A_1)$ the projection of the function on set A_1, i.e.

$$a \in A_1 \leftrightarrow f(a) \in f(A_1).$$

TODO: where do we need this?

For a few complexity arguments in this book we use big-Oh notation for comparing the asymptotic growth of functions $f, g : \mathbb{N} \to \mathbb{N}$.

$$f = O(g) \leftrightarrow \exists N_0 in \mathbb{N}, c > 0. \forall n \geq N_0. f(n) \leq c \cdot g(n)$$

3.1.2 Sequences, their indexing and overloading

We start this section with a remark on overloading. A mathematical symbol is *overloaded*, if it has different meanings in different contexts. A standard example is the addition symbol + in arithmetic expressions, which is for instance interpreted as integer addition in $1 + 2$ and as addition of fractions in $\frac{2}{3} + \frac{3}{4}$. Overloading is helpful to keep notation uncluttered, but it should of course only be used, where the meaning can be inferred from the context. Some situations - like $1 + \frac{2}{3}$ suggest several meanings of a symbol. In such cases one has to resolve the conflict by recoding one or more operands; in our example

$$1 + \frac{2}{3} = \frac{1}{1} + \frac{2}{3}$$

Finite sequences (resp. words or strings) a of n many elements a_i or $a[i]$ from a set A are a basic mathematical concept that is intuitively completely clear. If we try to formalize it, we find that there are several completely natural ways to do this. Let $n \geq 1$.

1. If we start indices with 1 and index from left to right we write

$$a = (a_1, \ldots, a_n) = a[1:n]$$

   which is formalized without three dots a a mapping

$$a : [1:n] \to A$$

   This is convenient for numbering symbols in a program text or statements in a statement sequence. With this formalization the set A^n of sequences of length n with elements from A is defined as

$$A^n = \{a | a : [1:n] \to A\}$$

2. If we start indices at 0 and index from left to right we write

$$a = (a_0, \ldots, a_{n-1}) = a[0:n-1]$$

   which is coded as

$$a : [0:n-1] \to A$$

   This is e.g. convenient for indexing nodes of paths in graphs. With this formalization the set A^n of sequences of length n with elements from A is defined as

$$A^n = \{a | a : [0:n-1] \to A\}$$

3. When dealing with number representations it turns out to be by far most convenient to start counting from 0 from right to left. We write

$$a = (a_{n-1}, \ldots, a_0) = a[n-1:0]$$

   which is also formalized as

$$a : [0:n-1] \to A$$

   and get again the formalization

$$A^n = \{a | a : [0:n-1] \to A\}$$

   Thus the direction of ordering does not show in the formalization *yet*. The reason is, that the interval $[0:n-1]$ is a set, and elements of sets are unordered.

This is changed when we define concatenation $a \circ b = c$ of sequences a and b. For the three cases we get

1.

$$a[1:n] \circ b[1:m] = c[1:n+m]$$

   with

$$c[i] = \begin{cases} a[i] & i \leq n \\ b[i-n] & i \geq n+1 \end{cases}$$

2.
$$a[0:n-1] \circ b[0:m-1] = c[0:n+m-1]$$
with
$$c[i] = \begin{cases} a[i] & i \leq n-1 \\ b[i-n] & i \geq n \end{cases}$$

3.
$$a[n-1:0] \circ b[m-1:0] = c[n+m-1:0]$$
with
$$c[i] = \begin{cases} b[i] & i \leq m-1 \\ a[i-m] & i \geq n \end{cases}$$

Concatenation of sequences a with single symbols $b \in A$ is handled by treating elements b as sequences with one element $b = b[1]$ if counting starts with 1 and $b = b[0]$ if counting starts with 0. The *empty sequence* ϵ is the unique sequence of length 0, thus one defines
$$A^0 = \{\epsilon\}$$
and it satisfies
$$a \circ \epsilon = \epsilon \circ a = a$$
in all formalizations. For any variant of indexing we define a to be a prefix of b if b has the form $a \circ c$:
$$prefix(a, b) \equiv \exists c.b = a \circ c$$

The set A^+ of nonempty finite sequences with elements from A is defined as
$$A^+ = \bigcup_{n \in \mathbb{N} \setminus \{0\}} A^n$$
and the set A^* of all finite sequences with elements from A as
$$A^* = \bigcup_{n \in \mathbb{N}} A^n = A^+ \cup \{\epsilon\}$$
Both definitions use A^n. Because A^n is overloaded, A^+ and A^* are overloaded too.

When forming subsequences, we get for the three cases

1. for $a[1:n]$ and $i \leq j$ and
$$a[i:j] = c[1:j-i+1] \quad \text{with} \quad c[k] = a[i+k-1]$$

2. for $a[0:n-1]$ and $i \leq j$
$$a[i:j] = c[0:j-i] \quad \text{with} \quad c[k] = a[i+k]$$

3. for $a[0:n-1]$ and $j \geq i$
$$a[j:i] = c[j-i:0] \quad \text{with} \quad c[k] = a[i+k]$$

In very few places we will have different indexing and direction in the same place, for instance if we identify in a program text $w[1:n]$ a substring $w[i:j]$ that we wish to interpret as a number representation $c[j-i:0]$ we have to do the trivial conversion
$$w[j:i] = c[j-i:0] \quad \text{with} \quad c[k] = a[i+k]$$

$\wedge$	and
$\vee$	or
$\neg$	not
$\oplus$	exclusive or, + modulo 2
$\rightarrow$	implies
$\leftrightarrow$	if and only if
$\forall$	for all
$\exists$	exists

Table 3.1: Logical connectives and quantifiers

3.1.3 Logical connectives and vector notation

For bits $x \in \mathbb{B}$ and natural numbers $n \in \mathbb{N}$ we denote the string obtained by repeating x exactly n times by x^n. In the form of an intelligence test:

$$x^n = \underbrace{x \ldots x}_{n \text{ times}}.$$

and in rigorous form

$$\begin{aligned} x^1 &= x \\ x^{n+1} &= x \circ x^n. \end{aligned}$$

where $\circ$ denotes the concatenation of bit strings. **Examples:** $1^2 = 11$ and $0^4 = 0000$.

For the concatenation of bit strings x_1 and x_2 we often omit $\circ$ and write

$$x_1 x_2 = x_1 \circ x_2.$$

In statements and predicates, we use the logical connectives and quantifiers from Table 3.1.3. For $\neg x$ we also write $\overline{x}$ or $/x$.

For $\circ \in \{\wedge, \vee, \oplus\}$, $a, b \in \mathbb{B}^n$ and a bit $c \in \mathbb{B}$, we borrow notation from vector calculus to define the corresponding bit-operations on bit-vectors:

$$\begin{aligned} \overline{a} &= (\overline{a_{n-1}}, \ldots, \overline{a_0}) \\ a[n-1:0] \circ b[n-1:0] &= (a_{n-1} \circ b_{n-1}, \ldots, a_0 \circ b_0) \\ c \circ b[n-1:0] &= (c \circ b_{n-1}, \ldots, c \circ b_0) \end{aligned}$$

Note that in the equations above, $\circ$ is not to be confused with the concatenation operator.

In computer science logarithms are to the base two unless explicitly stated otherwise. This text is no exception.

3.2 Modulo Computation

There are infinitely many integers and every computer can only store finitely many numbers. Thus, computer arithmetic cannot possibly work like ordinary arithmetic. Fixed point arithmetic[2] is usually performed modulo 2^n for some n. We review basics about modulo computation.

Definition 7 (Congruence Modulo)**.** *For integers $a, b \in \mathbb{Z}$ and natural numbers $k \in \mathbb{N}$ one defines a and b to be* congruent mod k *or* equivalent mod k *iff they differ by an integer multiple of k:*

$$a \equiv b \bmod k \leftrightarrow \exists z \in \mathbb{Z} : a - b = z \cdot k.$$

[2]The only arithmetic considered in this booklet. For the construction of floating point units see [MP00].

Definition 8 (Equivalence Relation)**.** *Let R be a relation between elements of a set A. We say that R is* reflexive *if we have aRa for all $a \in A$. We say that R is* symmetric *if aRb implies bRa. We say that R is* transitive *if aRb and bRc imply aRc. If all three properties hold, R is called an* equivalence relation *on A.*

An easy exercise shows

Lemma 7. *Congruence mod k is an equivalence relation.*

Proof. We show that the properties of an equivalence relation are satisfied:

- Reflexivity: For all $a \in \mathbb{Z}$ we have $a - a = 0 \cdot k$. Thus $a \equiv a \bmod k$ and congruence mod k is reflexive.
- Symmetry: Let $a \equiv b \bmod k$ with $a - b = z \cdot k$. Then $b - a = -z \cdot k$, thus $b \equiv a \bmod k$.
- Transitivity: Let $a \equiv b \bmod k$ with $a - b = z \cdot k$ and $b \equiv c \bmod k$ with $b - c = u \cdot k$. Then $a - c = (z + u) \cdot k$, thus $a \equiv c \bmod k$.

□

Lemma 8. *Let $a, b \in \mathbb{Z}$ and $k \in \mathbb{N}$ with $a \equiv a' \bmod k$ and $b \equiv b' \bmod k$. Then,*

$$\begin{aligned} a + b &\equiv a' + b' \bmod k \\ a - b &\equiv a' - b' \bmod k \\ a \cdot b &\equiv a' \cdot b' \bmod k. \end{aligned}$$

Proof. Let $a - a' = u \cdot k$ and $b - b' = v \cdot k$, then we have

$$\begin{aligned} a + b - (a' + b') &= a - a' + b - b' \\ &= (u + v) \cdot k \\ a - b - (a' - b') &= a - a' - (b - b') \\ &= (u - v) \cdot k \\ a \cdot b &= (a' + u \cdot k) \cdot (b' + v \cdot k) \\ &= a' \cdot b' + k \cdot (a' \cdot v + b' \cdot u + k \cdot u \cdot v) \end{aligned}$$

which imply the desired congruences. □

Two numbers r and s in an interval of the form $[i : i + k - 1]$ that are both equivalent to a mod k are identical:

Lemma 9. *Let $i \in \mathbb{Z}$, $k \in \mathbb{N}$, and let $r, s \in [i : i + k - 1]$, then*

$$a \equiv r \bmod k \wedge a \equiv s \bmod k \to r = s$$

Proof. By symmetry we have $s \equiv a \bmod k$ and by transitivity we get $s \equiv r \bmod k$. Thus $r - s = z \cdot k$ for an integer z. We conclude $z = 0$ because $|r - s| < k$. □

Definition 9 (System of Representatives)**.** *Let R be an equivalence relation on A. A subset $B \subset A$ is called a* system of representatives *if and only if for every $a \in A$ there is exactly one $r \in B$ with aRr. The unique $r \in B$ satisfying aRr is called the* representative *of a in B.*

Lemma 10. *For $i \in \mathbb{Z}$ and $k \in \mathbb{N}$, the interval of integers $[i : i + k - 1]$ is a system of representatives for equivalence mod k.*

Proof. Let $a \in \mathbb{Z}$. We define the representative $r(a)$ by

$$\begin{aligned} f(a) &= \begin{cases} max\{j \mid a - k \cdot j \geq i\} & a \geq i \\ min\{j \mid a + k \cdot j \geq i\} & a < i \end{cases} \\ r(a) &= \begin{cases} a - f(a) \cdot k & a \geq i \\ a + f(a) \cdot k & a < i. \end{cases} \end{aligned}$$

Then $r(a) \equiv a \bmod k$ and $r(a) \in [i : i + k - 1]$. Uniqueness follows from Lemma 9.

Note that, in case $i = 0$, $f(a)$ is the result of the integer division of a by k:

$$f(a) = \lfloor a/k \rfloor,$$

and

$$r(a) = a - \lfloor a/k \rfloor \cdot k$$

is the remainder of this division. □

We have to point out that in mathematics the three letter word 'mod' is not only used for the *relation* defined above. It is also used as a *binary operator* in which case $a \bmod k$ denotes the representative of a in $[0 : k - 1]$.

Definition 10 (Modulo Operator)**.** *For $a, b \in \mathbb{Z}$ and $k \in \mathbb{N}$,*

$$(a \bmod k) = \in\{b \mid a \equiv b \bmod k \wedge b \in [0 : k - 1]\}$$

Thus, $(a \bmod k)$ is the remainder of the integer division of a by k for $a \geq 0$. In order to stress when mod is used as a binary operator, we *always* write $(a \bmod k)$ in brackets. For later use in the theory of two's complement numbers we define another modulo operator:

Definition 11 (Two's Complement Modulo Operator)**.** *For $a, b \in \mathbb{Z}$ and an* even *number $k = 2 \cdot k'$ with $k' \in \mathbb{N}$,*

$$(a \operatorname{tmod} k) =\in \{b \mid a \equiv b \bmod k \wedge b \in [-k/2 : k/2 - 1]\}.$$

From Lemma 9 we infer a simple but useful lemma about the solution of equivalences mod k:

Lemma 11. *Let k be even and $x \equiv y \bmod k$ then*

1. $x \in [0 : k - 1] \rightarrow x = (y \bmod k)$,
2. $x \in [-k/2 : k/2 - 1] \rightarrow x = (y \operatorname{tmod} k)$.

3.3 Sums

3.3.1 Geometric sums

For $q \neq 1$ we consider

$$S = \sum_{i=0}^{n-1} q^i$$

the geometric sum over q. Then,

$$\begin{aligned} q \cdot S &= \sum_{i=1}^{n} q^i \\ q \cdot S - S &= q^n - 1 \\ S &= \frac{q^n - 1}{q - 1}. \end{aligned}$$

For $q = 2$ we get

Figure 3.1: Drawing an edge (u, v) from u to v

Lemma 12. *For $n \in \mathbb{N}$,*

$$\sum_{i=0}^{n-1} 2^i = 2^n - 1$$

which we will use in the next chapter.

3.3.2 Arithmetic Sums

For $n \in \mathbb{N}$ let

$$S_n = \sum_{i=^0}^{n} i$$

be the sum of the natural numbers from 0 to n. We recall that Gauß wrote $2 \cdot S_n$ as

$$\begin{aligned} 2 \cdot S_n &= 0 + 1 + \ldots + (n-1) + n \\ &\quad +n + (n-1) + \ldots + 1 + 0 \\ &= n \cdot (n+1) \end{aligned}$$

which gives

$$S_n = n \cdot (n+1)/2$$

If you are suspicious about proofs involving three dots (which you should), use the equation as an induction hypothesis and prove it by induction. While doing this do not define

$$\sum_{i=0}^{n} f_i = f_0 + \ldots + f_n$$

because that would involve three dots too. Instead define

$$\begin{aligned} \sum_{i=0}^{0} f_i &= f_0 \\ \sum_{i=0}^{n} f_i &= (\sum_{i=0}^{n-1} f_i) + f_n \end{aligned}$$

3.4 Graphs

3.4.1 Directed Graphs

In graph theory a *directed graph* G is specified by

- a set $G.V$ of nodes. Here we consider only finite graphs, thus $G.V$ is finite.
- a set $G.E \subset G.V \times G.V$ of edges. Edges $(u, v) \in G.E$ are depicted as arrows from node u to node v as shown in figure 3.1. For $(u, v) \in G.E$ one says that v is a successor of u and that u is a predecessor of v.

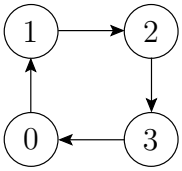


Figure 3.2: Graph G

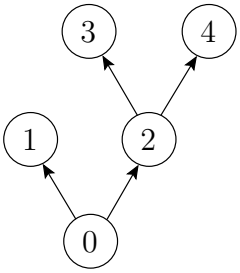


Figure 3.3: Graph G'

If it is clear which graph G is meant, one abbreviates

$$\begin{aligned} V &= G.V \\ E &= G.E \end{aligned}$$

The graph G in figure 3.2 is formally described by

$$\begin{aligned} G.V &= \{0,1,2,3\} \\ G.E &= \{(0,1),(1,2),(2,3),(3,0)\} \end{aligned}$$

The graph G' in figure 3.3 is formally described by

$$\begin{aligned} G'.V &= \{0,1,2,3,4\} \\ G'.E &= \{(0,1),(0,2),(2,3),(2,4)\} \end{aligned}$$

Let $G = (V, E)$ be a directed graph. For nodes $u, v \in V$ a *path* from u to v in G is a sequence $p[0:k]$ of nodes $p[i] \in V$, that begins with u, ends with v and in which subsequent elements are connected by edges

$$\begin{aligned} p_0 &= u \\ p_k &= v \\ i > 0 &\rightarrow (p_{i-1}, p_i) \in E \end{aligned}$$

The number k is called the *length* of the path. We denote it by $le(p)$. A path from u to u with length at least 1 is called a *cycle*. In figure 3.2 the sequence of nodes $(0,1,2,3,0,1)$ is a path of length 5 from node 0 to node 1, and the sequence $(1,2,3,0,1)$ is a cycle. In figure 3.3 the sequence of nodes $(0,2,4)$ is a path of length 2 from node 0 to node 4.

For graphs G and nodes $v \in G.V$ the indegree $indeg(v, G)$ of node v in graph G is defined as the number of edges ending in v and the outdegree $outdeg(v, G)$ of node v in graph G is defined as the number of edges starting in v

$$\begin{aligned} indeg(v,G) &= \#\{u : (u,v) \in G.E\} \\ outdeg(v,G) &= \#\{x : (v,x) \in G.E\} \end{aligned}$$

In the graph of figure 3.2 we have

$$indeg(v,G) = outdeg(v,G) = 1$$

for all nodes v. In the graph G' of figure 3.3 we have

$$\begin{aligned} & indeg(0,G') = 0 \\ v \neq 0 &\rightarrow indeg(v,G') = 1 \\ v \in \{1,3,4\} &\rightarrow outdeg(v,G') = 0 \\ v \in \{0,2\} &\rightarrow outdeg(v,G') = 2 \end{aligned}$$

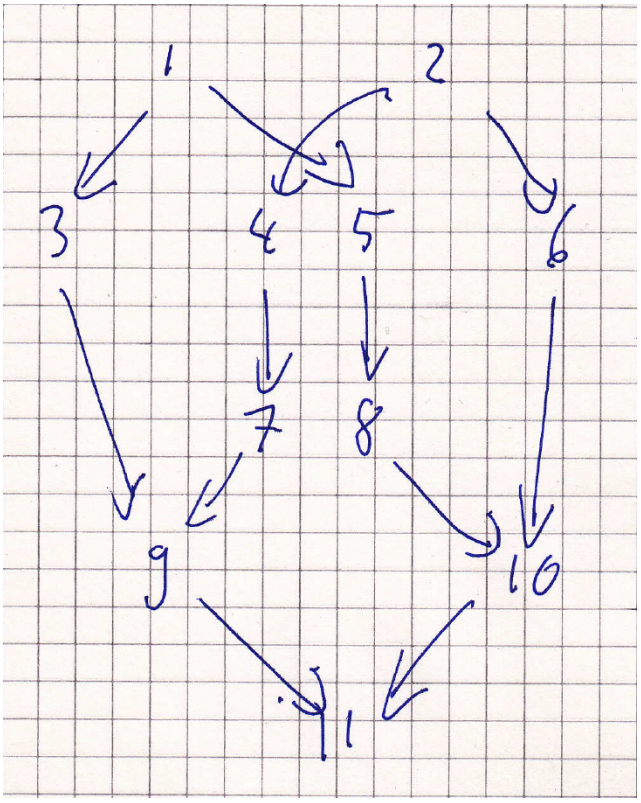


Figure 3.4: Example of a directed acylic graph

A source of a graph G is a node v with $indeg(v, G) = 0$ and a sink of a graph is a node v with $outdeg(v, G) = 0$. The graph in figure 3.2 has neither sources nor sinks. The graph in figure 3.3 has a single source $v = 0$ and sinks 1,3 and 4. If it is clear which graph is meant one abbreviates

$$\begin{aligned} indeg(v) &= indeg(v, G) \\ outdeg(v) &= outdeg(v, G) \end{aligned}$$

3.4.2 Directed acyclic graphs and the depth of nodes

A *directed acyclic graph* or DAG is simply a directed graph without cycles. For the remainder of this subsection we will use the graph from figure 3.4 an a running example.

Lemma 13. *In directed acyclic graphs $G = (V, E)$ the length $le(p)$ of paths p is bounded by the number of nodes minus 1.*

$$le(p) \leq \#V - 1$$

Proof. Assume you have n pigeons sitting in $n - 1$ holes. Then there must be two pigeons which are sitting in the same hole, because otherwise there would be at most $n - 1$ pigeons. We use this so called *pigeon hole argument* to prove the lemma by contradiction.

Let $n = \#V$ be the number of nodes and assume that $p[0 : n]$ is a path in G. We treat the indices $i \in [0 : n]$ as pigeons, the vertices $v \in V$ as pigeon holes, and pigen i sits in hole $p(i)$. We conclude, that there must be two different indices i and j such that $p(i) = p(j)$. But then $p[i : j]$ is a cycle and G is not acyclic. □

In the sequel this simple lemma will turn out to be amazingly useful.

Lemma 14. *Let $G = (V, E)$ be a directed acyclic graph. Then G has at least one source and one sink.*

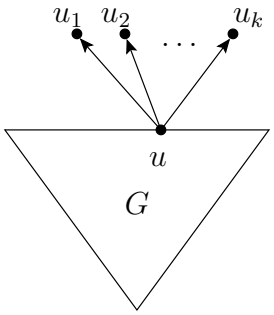


Figure 3.5: Generating a rooted tree G' by adding new nodes $u_1, \ldots, u_k$ as successors of a sink u of rooted tree G

Proof. Let $n = \#V$. Pick any node $v \in V$ in the graph. Starting from v follow edges forward repeatedly as long as they are present. By lemma 13 one hits a sink after at most $n - 1$ edges. Following edges backward one finds a source. □

In the graph in figure 3.4 we end in sink 11 when we follow edges forward no matter where we start; this is not surprising because there is only one sink. Following edges backward we arrive in source 1 if we start in node 3, in source 2 if we start in node 7 and in source 1 or 2 if we start in node 9, depending which of the edges $(7, 9)$ and $(3, 9)$ we follow.

The previous two lemmas allow to define the *depth* $de(v, G)$ of a node v in a DAG G as the length of a longest path in G from a source to v. If it is clear what graph is meant one drops the argument G and abbreviates

$$de(v) = de(v, G)$$

A simple argument shows that the depth of sources is 0 and that the depth on node v is one greater than the depth of one of its predecessors

$$de(v, G) = \begin{cases} 0 & v \quad \text{is a source} \\ max\{de(u, v) : (u, v) \in \text{G.E}\} + 1 & \text{otherwise} \end{cases}$$

Thus in figure x we have

$$de(v) = \begin{cases} 1 & v \in \{3, 4, 5, 6\} \\ 2 & v \in \{7, 8\} \\ 3 & v \in \{9, 10\} \\ 4 & v = 11 \end{cases}$$

The depth of a DAG is the maximum of the depth of its nodes

$$de(G) = max\{de(v, G) : v \in G.V\}$$

Thus the graph in figure x has depth 4.

3.4.3 Rooted Trees

We define rooted trees in the following way

1. a graph with a single node and no edges is a rooted tree.
2. if G is a rooted tree, u is a sink of G and $v_0, \ldots, v_{k-1} \notin G.V$ are new nodes, then the graph G' defined by

$$\begin{aligned} G'.V &= G.V \cup \{v_0, \ldots, v_{k-1}\} \\ G'.E &= G.E \cup \{(u, v_0), \ldots, (u, v_{k-1})\} \end{aligned}$$

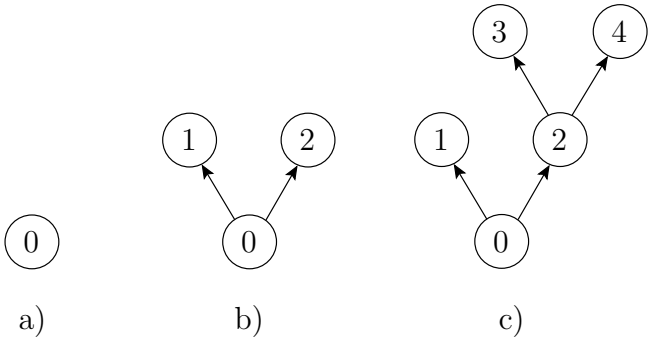


Figure 3.6: Generating a rooted tree G' by adding new nodes $u_1, \ldots, u_k$ as successors of a sink u of rooted tree G

is a rooted trees (see figure 3.5).

3. all rooted trees can be generated in finitely many steps by the above two rules.

By this definition the graph in figure 3.3 is a rooted tree, because it can be generated in 3 steps as shown in figure 3.6 a) to c). Rooted trees have a single source which is called the *root* of the tree. The sinks of trees are called *leaves* of the tree. If (u, v) is an edge of a tree one calls u the *father* of v and one calls v a *son* of u.

The following series of easy exercises relates the above recursive definition of rooted trees to a common definition of rooted trees in graph theory.

Exercises

1. Show that every rooted tree has exactly one source (called the root) and that for every node u of a rooted tree there is exactly one path from the root to u.

2. Now assume that G is a finite directed graph with a single source r (called the root) and such that for every node $u \in G.V$ there is exactly one path from r to u in G. Let $n = \#G.V$ be the number of nodes of G

   (a) show that every path in G has length at most n. Hint: use a pigeon hole argument

   (b) construct a path $p[0 : y]$ in G' in the following way: i) initially $y = 0$ and $p[0] = r$. ii) if all successors v_i of $p(y)$ are sinks, set $u = p(y)$ and stop. Otherwise extend the path by choosing $p(y + 1)$ among the successors of $p(y)$ that are not a sink. Show that we stop with some $y \leq n$.

   (c) show that there is a node u in G such that all successors of u are sinks

   (d) show that a finite graph is a rooted tree iff it has a single source r and such that for every node u in G there is exactly one path from r to u

Chapter 4

Number Formats and Boolean Algebra

Section 4.1 introduces the binary number format, presents the school method for binary addition and proves that it works. Although this will look completely familiar and the correctness proof of the addition algorithms is only a few lines long, the reader should treat this result with deep respect: it is probably the first time that he or she sees a proof of the fact that the addition algorithm he learned at school always works. The Old Romans, who were fabulous engineers in spite of their clumsy number systems, would have *loved* to see this proof.

Integers are represented in computers as two's complement numbers. In Section 4.2 we introduce this number format and derive a small number of basic identities for such numbers. From this we derive a subtraction algorithms for binary numbers, which is quite different from the school method, and show that it works. Sections 3.2, **??** and 4.2 are the basis of our constructions of an arithmetic unit later.

Finally, in Section 4.3 on Boolean Algebra, we provide a very short proof of the fundamental result that boolean functions can be computed using boolean expressions in disjunctive normal form. This result can serve to construct all small circuits - e.g. in the control logic - where we only specify their functionality and do not bother to specify a concrete realization. The proof is intuitive and looks simple, but it will give us the occasion to explain formally the difference between what is often called "two kinds of equations": i) identities[1] $e(x) = e'(x)$ which hold for all x and ii) equations $e(x) = e'(x)$ that we want to solve by determining the set of all x such that the equation holds[2]. The reader will notice that this might be slightly subtle, because both kinds of equations have exactly the same form.

4.1 Binary Numbers

Definition 12 (Binary Number Interpretation)**.** *For bit-strings $a = a[n-1:0] \in \mathbb{B}^n$ we denote by*

$$\langle a \rangle = \sum_{i=0}^{n-1} a_i \cdot 2^i$$

the interpretation of bit-string a as a binary number*. We call a the* binary representation *of the natural number $\langle a \rangle$.*

Examples:

$$\begin{aligned} \langle 100 \rangle &= 4 \\ \langle 111 \rangle &= 7 \\ \langle 10^n \rangle &= 2^n. \end{aligned}$$

[1] In German: Identitäten.

[2] In German: Bestimmungsgleichung.

Applying Lemma 12, we get

$$\langle 1^n \rangle = \sum_{i=0}^{n-1} 2^i = 2^n - 1,$$

i.e., the largest binary number representable with n bits corresponds to the natural number $2^n - 1$.

Note that binary number interpretation is an injective function:

Lemma 15. *Let $a, b \in \mathbb{B}^n$. Then,*

$$a \neq b \rightarrow \langle a \rangle \neq \langle b \rangle.$$

Proof. Let $j = \max\{i \mid a_i \neq b_i\}$ be the largest index where strings a and b differ. Without loss of generality assume $a_j = 1$ and $b_j = 0$. Then

$$\begin{aligned} \langle a \rangle - \langle b \rangle &= \sum_{i=0}^{j} a_i \cdot 2^i - \sum_{i=0}^{j} b_i \cdot 2^i \\ &\geq 2^j - \sum_{i=0}^{j-1} 2^i \\ &= 1 \end{aligned}$$

by Lemma 12. □

Definition 13. *Let $n \in \mathbb{N}$. We denote by*

$$B_n = \{\langle a \rangle \mid a \in \mathbb{B}^n\}$$

the set of natural numbers that have a binary representation of length n.

Since

$$0 \leq \langle a \rangle \leq \sum_{i=0}^{n-1} 2^i = 2^n - 1$$

we deduce

$$B_n \subseteq [0 : 2^n - 1].$$

As $\langle \cdot \rangle$ is injective and

$$\#B_n = \#[0 : 2^n - 1] = 2^n$$

we observe that $\langle \cdot \rangle$ is bijective and, thus, we have

Lemma 16. *For $n \in \mathbb{N}$, we have*

$$B_n = [0 : 2^n - 1]$$

Definition 14 (Binary Representation)**.** *For $x \in B_n$ we denote the binary representation of x of length n by $bin_n(x)$:*

$$bin_n(x) = \in\{a \mid a \in \mathbb{B}^n \wedge \langle a \rangle = x\}.$$

To shorten notation even further, we write x_n instead of $bin_n(x)$:

$$x_n = bin_n(x).$$

It is often useful to decompose n bit binary representations $a[n-1:0]$ into an upper part $a[n-1:m]$ and a lower part $a[m-1:0]$. The connection between the numbers represented is stated in

a	b	c	c'	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Table 4.1: Binary addition of 1 bit numbers a, b with carry c

Lemma 17 (Decomposition Lemma)**.** *Let $a \in \mathbb{B}^n$ and $n \geq m$. Then,*

$$\langle a[n-1:0]\rangle = \langle a[n-1:m]\rangle \cdot 2^m + \langle a[m-1:0]\rangle.$$

Proof.

$$\begin{aligned}
\langle a[n-1:0]\rangle &= \sum_{i=m}^{n-1} a_i \cdot 2^i + \sum_{i=0}^{m-1} a_i \cdot 2^i \\
&= \sum_{j=0}^{n-1-m} a_{m+j} \cdot 2^{m+j} + \langle a[m-1:0]\rangle \\
&= 2^m \cdot \sum_{j=0}^{n-1-m} a_{m+j} \cdot 2^j + \langle a[m-1:0]\rangle \\
&= 2^m \cdot \langle a[n-1:m]\rangle + \langle a[m-1:0]\rangle
\end{aligned}$$

□

We obviously have

$$\langle a[n-1:0]\rangle \equiv \langle a[m-1:0]\rangle \bmod 2^m.$$

Using Lemma 11, we infer

Lemma 18. *For $a \in \mathbb{B}^n$ and $m \leq n$,*

$$\langle a[m-1:0]\rangle = (\langle a[n-1:0]\rangle \bmod 2^m)$$

Intuitively speaking, taking a binary number modulo 2^m means 'throwing away' the bits with position m or higher.

Table 4.1 specifies the addition algorithm for binary numbers a, b of length 1 and a carry-bit c. The binary representation $(c', s) \in \mathbb{B}^2$ of the sum of bits $a, b, c \in \mathbb{B}$ is computed as

$$\langle c's\rangle = a + b + c.$$

For the addition of n bit numbers $a[n-1:0]$ and $b[n-1:0]$ with carry in c_0 we first observe for the sum S:

$$\begin{aligned}
S &= \langle a[n-1:0]\rangle + \langle b[n-1:0]\rangle + c_0 \\
&\leq 2^n - 1 + 2^n - 1 + 1 \\
&= 2^{n+1} - 1.
\end{aligned}$$

Thus, the sum $S \in \mathbb{B}^{n+1}$ can be represented as a binary number $\langle s[n:0]\rangle$ with $n+1$ bits. For the computation of the sum bits we use the method for long addition that we learn in elementary school for decimal numbers. We denote by c_i the carry from position $i-1$ to position i and compute (c_{i+1}, s_i) using the basic binary addition of Table 4.1:

$$\begin{aligned} \langle c_{i+1}, s_i\rangle &= a_i + b_i + c_i \\ s_n &= c_n. \end{aligned} \tag{4.1}$$

That this algorithm indeed computes the sum of the input numbers is asserted in

Lemma 19 (Correctness of the Binary Addition Algorithm)**.** *Let $a, b \in \mathbb{B}^n$ and let $c \in \mathbb{B}$. Further, let $c_n \in \mathbb{B}$ and $s \in \mathbb{B}^n$ be computed according to the addition algorithm described above. Then,*

$$\langle c_n, s[n-1:0]\rangle = \langle a[n-1:0]\rangle + \langle b[n-1:0]\rangle + c_0$$

Proof. By induction on n. For $n = 0$ this follows directly from Equation 4.1. For the induction step we conclude from $n-1$ to n:

$$\begin{aligned} & \langle a[n-1:0]\rangle + \langle b[n-1:0]\rangle + c_0 \\ = \;& (a_{n-1} + b_{n-1}) \cdot 2^{n-1} + \langle a[n-2:0]\rangle + \langle b[n-2:0]\rangle + c_0 \\ = \;& (a_{n-1} + b_{n-1}) \cdot 2^{n-1} + \langle c_{n-1}, s[n-2:0]\rangle \qquad \text{(induction hypothesis)} \\ = \;& (a_{n-1} + b_{n-1} + c_{n-1}) \cdot 2^{n-1} + \langle s[n-2:0]\rangle \\ = \;& \langle c_n, s_{n-1}\rangle \cdot 2^{n-1} + \langle s[n-2:0]\rangle \qquad \text{(Equation 4.1)} \\ = \;& \langle c_n, s[n-1:0].\rangle \qquad \text{(Lemma 17)} \end{aligned}$$

□

The following simple lemma allows to break the addition of two long numbers into two additions of shorter numbers. It is useful amazingly useful. In this text we will use it in subsection 7.1.2 for showing the correctness of recursive addition algorithms as well as in subsection 8.3.8 for justifying the addition 32 bit numbers with 30-bit adders if the binary representation of one of the numbers ends with 00. In the hardware oriented text [**?**] it is also used to show the correctness of the shifter constructions supporting aligned byte and half word accesses in cache lines.

Lemma 20 (Decomposition of Binary Number Addition)**.** *For $a, b \in \mathbb{B}^n$, for $d, e \in \mathbb{B}^m$ and for $c_0, c', c'' \in \mathbb{B}$ let*

$$\begin{aligned} \langle d\rangle + \langle e\rangle + c_0 &= \langle c't[m-1:0]\rangle \\ \langle a\rangle + \langle b\rangle + c' &= \langle c''s[n-1:0]\rangle, \end{aligned}$$

then

$$\langle ad\rangle + \langle be\rangle + c_0 = \langle c''st\rangle.$$

Repeatedly using Lemma 17, we have

$$\begin{aligned} & \langle ad\rangle + \langle be\rangle + c_0 \\ = \;& \langle a\rangle \cdot 2^m + \langle d\rangle + \langle b\rangle \cdot 2^m + \langle e\rangle + c_0 \\ = \;& (\langle a\rangle + \langle b\rangle) \cdot 2^m + \langle c't\rangle \\ = \;& (\langle a\rangle + \langle b\rangle + c') \cdot 2^m + \langle t\rangle \\ = \;& \langle c''s\rangle \cdot 2^m + \langle t\rangle \\ = \;& \langle c''st\rangle. \end{aligned}$$

4.2 Two's Complement Numbers

Definition 15 (Two's Complement Number Interpretation)**.** *For bit-strings $a[n-1:0] \in \mathbb{B}^n$ we denote by*

$$[a] = -a_{n-1} \cdot 2^{n-1} + \langle a[n-2:0] \rangle$$

the interpretation of a as a two's complement number*. We refer to a as the* two's complement representation *of $[a]$.*

Definition 16. *For $n \in \mathbb{N}$, we denote by*

$$T_n = \{[a] \mid a \in \mathbb{B}^n\}$$

the set of integers that have a two's complement representation of length n.

Since

$$\begin{aligned} T_n &= \{[0b] \mid b \in \mathbb{B}^{n-1}\} \cup \{[1b] \mid b \in \mathbb{B}^{n-1}\} \\ &= B_{n-1} \cup \{-2^{n-1} + x \mid x \in B_{n-1}\} \\ &= [0 : 2^{n-1} - 1] \cup \{-2^{n-1} + x \mid x \in [0 : 2^{n-1} - 1]\} \qquad \text{(Lemma 16)} \end{aligned}$$

we have

Lemma 21. *Let $n \in \mathbb{N}$. Then,*

$$T_n = [-2^{n-1} : 2^{n-1} - 1]$$

Definition 17 (Two's Complement Representation)**.** *By $twoc_n(x)$ we denote the two's complement representation of $x \in T_n$:*

$$twoc_n(x) = \in \{a \mid a \in \mathbb{B}^n \wedge [a] = x\}.$$

Basic properties of two's complement numbers are summarized in

Lemma 22. *Let $a \in \mathbb{B}^n$. Then, the following hold:*

$$\begin{aligned} [0a] &= \langle a \rangle \quad \textit{(embedding)} \\ [a] &\equiv \langle a \rangle \bmod 2^n \\ [a] < 0 &\leftrightarrow a_{n-1} = 1 \quad \textit{(sign bit)} \\ [a_{n-1}a] &= [a] \quad \textit{(sign extension)} \\ -[a] &= [\overline{a}] + 1. \end{aligned}$$

Proof. The first line is trivial. The second line follows from

$$\begin{aligned} [a] - \langle a \rangle &= -a_{n-1} \cdot 2^{n-1} + \langle a[n-2:0] \rangle - (a_{n-1} \cdot 2^{n-1} + \langle a[n-2:0] \rangle) \\ &= -a_{n-1} \cdot 2^n. \end{aligned}$$

If $a_{n-1} = 0$ we have $[a] = \langle a[n-2:0] \rangle \geq 0$. If $a_{n-1} = 1$ we have

$$\begin{aligned} [a] &= -2^{n-1} + \langle a[n-2:0] \rangle \\ &\leq -2^{n-1} + 2^{n-1} - 1 \quad \text{(Lemma 16)} \\ &= -1. \end{aligned}$$

This shows the third line. The fourth line follows from

$$\begin{aligned} [a_{n-1}a] &= -a_{n-1} \cdot 2^n + \langle a[n-1:0] \rangle \\ &= -a_{n-1} \cdot 2^n + a_{n-1} \cdot 2^{n-1} + \langle a[n-2:0] \rangle \\ &= -a_{n-1} \cdot 2^{n-1} + \langle a[n-2:0] \rangle \\ &= [a]. \end{aligned}$$

x	y	$\overline{x}$	$x \wedge y$	$x \vee y$	$x \oplus y$
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

Table 4.2: Boolean operators

For the last line we observe that $\overline{x} = 1 - x$ for $x \in \mathbb{B}$. Then

$$\begin{aligned}
[\overline{a}] &= -\overline{a_{n-1}} \cdot 2^{n-1} + \sum_{i=0}^{n-2} \overline{a_i} \cdot 2^i \\
&= -(1 - a_{n-1}) \cdot 2^{n-1} + \sum_{i=0}^{n-2} (1 - a_i) \cdot 2^i \\
&= -2^{n-1} + \sum_{i=0}^{n-2} 2^i + a_{n-1} \cdot 2^{n-1} - \sum_{i=0}^{n-2} a_i \cdot 2^i \\
&= -1 - [a]. \quad \text{(Lemma 12)}
\end{aligned}$$

□

We conclude the discussion of binary numbers and two's complement numbers with a lemma that provides a subtraction algorithm for binary numbers:

Lemma 23. *Let $a, b \in \mathbb{B}^n$. Then*

$$\langle a \rangle - \langle b \rangle \equiv \langle a \rangle + \langle \overline{b} \rangle + 1 \bmod 2^n.$$

If additionally $\langle a \rangle - \langle b \rangle \geq 0$, we have

$$\langle a \rangle - \langle b \rangle = (\langle a \rangle + \langle \overline{b} \rangle + 1 \bmod 2^n).$$

Proof. By Lemma 22 we have

$$\begin{aligned}
\langle a \rangle - \langle b \rangle &= \langle a \rangle - [0b] \\
&= \langle a \rangle + [1\overline{b}] + 1 \\
&= \langle a \rangle - 2^n + \langle \overline{b} \rangle + 1 \\
&\equiv \langle a \rangle + \langle \overline{b} \rangle + 1 \bmod 2^n.
\end{aligned}$$

The extra hypothesis $\langle a \rangle - \langle b \rangle \geq 0$ implies

$$\langle a \rangle - \langle b \rangle \in B_n.$$

The second claim now follows from Lemma 11 □

4.3 Boolean Algebra

We consider Boolean expressions with constants 0 and 1, variables $x_0, x_1, \ldots$, $a, b, \ldots$, and function symbols $^-, \wedge, \vee, \oplus, f(\ldots), g(\ldots), \ldots$. Four of the function symbols have predefined semantics as specified in Table 4.2. For a more formal definition one collects the constants, variables and function

symbols allowed into sets

$$\begin{aligned} C &= \{0,1\} \\ V &= \{x_0, x_1, \ldots\} \\ F &= \{f_0, f_1, \ldots\} \end{aligned}$$

and denotes the number of arguments for function f_i with n_i. Now we can define the set BE of Boolean expressions by the following rules:

1. constants and variables are Boolean expressions

$$C \cup V \subset BE$$

2. if e is a Boolean expression, then also $(\overline{e})$

$$e \in BE \rightarrow (\overline{e}) \in BE$$

3. if e and e' are boolean expresions then so is $(e \circ e')$ for any of the binare predefined connectives $\circ$

$$e, e' \in BE \wedge \circ \in \{\wedge, \vee, \oplus\} \rightarrow (e \circ e') \in BE$$

4. if f_i is a symbol for a function with n_i arguments, then we can obtain a Boolean expression $f_i(e_1, \ldots, e_{n_i})$ by substituting Boolean expressions e_j

$$(\forall j \in [1 : n_i] : e_j \in BE) \rightarrow f_i(e_1, \ldots, e_{n_i}) \in BE$$

5. all Boolean expressions are formed by the above rules.

We call a Boolean expression *pure* if it uses only the predefined connectives.

In order to save brackets one uses the convention, that $\overline{\ }$ binds stronger than $\wedge$ and that $\wedge$ binds stronger than $\vee$. Thus $\overline{x_1} \wedge x_2 \vee x_3$ is an abbreviation for

$$\overline{x_1} \wedge x_2 \vee x_3 = ((\overline{x_1}) \wedge x_2) \vee x_3.$$

We denote expressions e depending on variables $x = x[1 : n]$ often by $e(x)$. Variables x_i can take values in $\mathbb{B}$, thus $x = x[1 : n]$ can take values in $\mathbb{B}^n$. We denote the result of evaluation expression $e \in BE$ with bit string $a \in \mathbb{B}^n$ by $e(a)$ and get a straight forward set of rules for evaluating expressions:

1. Substitute a_i for x_i

$$x_i(a) = a_i$$

2. evaluate $(\overline{e})$ by evaluating e and negating the result according to the predefined meaning of negation in table 4.2

$$(\overline{e})(a) = \overline{e(a)}$$

3. evaluate $(e \circ e')$ by evaluating e and e' and then combining the results according to the predefined meaning of $\circ$ in table 4.2

$$(e \circ e')(a) = e(a) \circ e'(a)$$

4. expressions of the form $f_i(e_1, \ldots, e_{n_j})$ can only be evaluated if symbol f_i has an interpretation as a function

$$f_i : \mathbb{B}^{n_i} \to \mathbb{B}$$

In this case evaluate $f_i(e_1, \ldots, e_{n_j})(a)$ by evaluating arguments e_j, substituting the result into f and evaluating f.

$$f_i(e_1, \ldots, e_{n_j})(a) = f_i(e_1(a), \ldots, e_{n_i}(a))$$

The following small example shows that this very formal and detailed set of rules captures our usual way of evaluating expressions.

$$\begin{aligned}(x_1 \wedge x_2)(0,1) &= x_1(0,1) \wedge x_2(0,1) \\ &= 0 \wedge 1 \\ &= 0\end{aligned}$$

Equations have thefor

$$e = e'$$

where e and e' are expressions (involving here variables $x = x[1:n]$. They come in two flavors:

- identities. An equation $e = e'$ is an identity iff expressions e and e' evaluate to the same value $\in \mathbb{B}$ for any substitution of the variables $a = a[1:n] \in \mathbb{B}^n$:

$$\forall a \in \mathbb{B}^n : e(a) = e'(a),$$

- equations which one wants to solve. A substitution $a = a[1:n] \in \mathbb{B}^n$ solves equation $e = e'$ if $e(a) = e'(a)$.

If not stated otherwise, we usually assume equations to be of the first type, i.e. to be implicitly quantified over all free variables. This is also the case with *definitions* of functions, where the left-hand side of an equation represents an entity being defined. For instance, the following definition of the function

$$f(x_1, x_2) = x_1 \wedge x_2$$

is the same as

$$\forall a, b \in \mathbb{B} : f(a, b) = a \wedge b.$$

We may also write

$$e \equiv e'$$

to stress that a given equation is an identity or to avoid brackets in case if this equation is a definition and the right-hand side itself contains an equality sign.

In case we talk about several equations in a single statement, we assume implicit quantification over the whole statement rather than over every single equation. In that case we usually have equations of the second kind. For instance

$$e_1 = e_2 \leftrightarrow e_3 = 0$$

is the same as

$$\forall a \in \mathbb{B}^n : e_1(a) = e_2(a) \leftrightarrow e_3(a) = 0$$

and means that, for a given substitution a, equations e_1 and e_2 evaluate to the same value iff equation e_3 evaluates to 0.

In Boolean algebra there is a very simple connection between the solution of equations and identities. An identity $e \equiv e'$ holds iff equations $e = 1$ and $e' = 1$ have the same set of solutions.

Lemma 24. *Given Boolean expressions $e(x)$ and $e'(x)$, we have*

$$e \equiv e' \leftrightarrow \forall a \in \mathbb{B}^n : (e = 1 \leftrightarrow e' = 1)$$

Proof. The direction from left to right is trivial. For the other direction we distinguish cases:

- $e(a) = 1$. Then $e'(a) = 1$ by hypothesis,
- $e(a) = 0$. Then $e'(a) = 1$ would by hypothesis imply the contradiction $e(a) = 1$. Because in Boolean algebra $e'(a) \in \mathbb{B}$ we conclude $e'(a) = 0$.

Thus, we have $e(a) = e'(a)$ for all $a \in \mathbb{B}^n$. □

4.3.1 Identities

In the following, we provide a list of useful identities of Boolean algebra.

- Commutativity:

$$\begin{aligned} x_1 \wedge x_2 &\equiv x_2 \wedge x_1 \\ x_1 \vee x_2 &\equiv x_2 \vee x_1 \\ x_1 \oplus x_2 &\equiv x_2 \oplus x_1 \end{aligned}$$

- Associativity:

$$\begin{aligned} (x_1 \wedge x_2) \wedge x_3 &\equiv x_1 \wedge (x_2 \wedge x_3) \\ (x_1 \vee x_2) \vee x_3 &\equiv x_1 \vee (x_2 \vee x_3) \\ (x_1 \oplus x_2) \oplus x_3 &\equiv x_1 \oplus (x_2 \oplus x_3) \end{aligned}$$

- Distributivity:

$$\begin{aligned} x_1 \wedge (x_2 \vee x_3) &\equiv (x_1 \wedge x_2) \vee (x_1 \wedge x_3) \\ x_1 \vee (x_2 \wedge x_3) &\equiv (x_1 \vee x_2) \wedge (x_1 \vee x_3) \end{aligned}$$

- Identity:

$$\begin{aligned} x_1 \wedge 1 &\equiv x_1 \\ x_1 \vee 0 &\equiv x_1 \end{aligned}$$

- Idempotence:

$$\begin{aligned} x_1 \wedge x_1 &\equiv x_1 \\ x_1 \vee x_1 &\equiv x_1 \end{aligned}$$

- Annihilation:

$$\begin{aligned} x_1 \wedge 0 &\equiv 0 \\ x_1 \vee 1 &\equiv 1 \end{aligned}$$

x_1	x_2	$x_1 \wedge x_2$	$\overline{x_1 \wedge x_2}$	$\overline{x_1}$	$\overline{x_2}$	$\overline{x_1} \vee \overline{x_2}$
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0

Table 4.3: Verifying the first of de Morgan's laws

- Absorption:

$$\begin{aligned} x_1 \vee (x_1 \wedge x_2) &\equiv x_1 \\ x_1 \wedge (x_1 \vee x_2) &\equiv x_1 \end{aligned}$$

- Complement:

$$\begin{aligned} x_1 \wedge \overline{x_1} &\equiv 0 \\ x_1 \vee \overline{x_1} &\equiv 1 \end{aligned}$$

- Double negation:

$$\overline{\overline{x_1}} \equiv x_1$$

- De Morgan's laws:

$$\begin{aligned} \overline{x_1 \wedge x_2} &\equiv \overline{x_1} \vee \overline{x_2} \\ \overline{x_1 \vee x_2} &\equiv \overline{x_1} \wedge \overline{x_2}. \end{aligned}$$

Each of these identities can be proven in a simple brute force way: if the identity has n variables, then for each of the 2^n possible substitutions of the variables the left and right hand sides of the identities are evaluated with the help of Table 4.2. If for each substitution the left hand side and the right hand side evaluate to the same value, then the identity holds. For the first of de Morgan's laws this is illustrated in Table 4.3.

4.3.2 Solving Equations

We consider expressions e and e_i (where $1 \leq i \leq n$), involving a vector of variables x. We derive three basic lemmas about the solution of Boolean equations. For $a \in \mathbb{B}$ we define

$$e^a = \begin{cases} e & a = 1 \\ \overline{e} & a = 0 \end{cases}.$$

Inspection of the semantics of $^-$ in Table 4.2 immediately gives

Lemma 25 (Solving Negation)**.** *Given a Boolean expression $e(x)$ and $a \in \mathbb{B}$, we have*

$$e^a = 1 \leftrightarrow e = a$$

Inspection of the semantics of $\wedge$ in Table 4.2 gives

$$(e_1 \wedge e_2) = 1 \leftrightarrow e_1 = 1 \wedge e_2 = 1.$$

Induction on n results in

Lemma 26 (Solving Conjunction)**.** *Given Boolean expressions $e_i(x)$, where $1 \leq i \leq n$, we have*

$$(\bigwedge_{i=1}^{n} e_i) = 1 \leftrightarrow \forall i \in [1:n] : e_i = 1$$

From the semantics of $\vee$ in Table 4.2, we have

$$(e_1 \vee e_2) = 1 \leftrightarrow e_1 = 1 \vee e_2 = 1.$$

Induction on n yields

Lemma 27 (Solving Disjunction)**.** *Given Boolean expressions e_i, where $1 \leq i \leq n$, we have*

$$(\bigvee_{i=1}^{n} e_i) = 1 \leftrightarrow \exists i \in [1:n] : e_i = 1$$

4.3.3 Disjunctive Normal Form

Definition 18. *Let $f : \mathbb{B}^n \to \mathbb{B}$ be a switching function*[3] *and let e be a Boolean expression with variables x. We say that e* computes *f iff the identity $f(x) \equiv e$ holds.*

Lemma 28. *Every switching function is computed by some Boolean expression:*

$$\forall f : \mathbb{B}^n \to \mathbb{B} \ : \exists e : f(x) \equiv e$$

Moreover expression e is pure.

Proof. Let $b \in \mathbb{B}$ and let x_i be a variable. We define the *literal*

$$x_i^b = \begin{cases} x_i & b = 1 \\ \overline{x_i} & b = 0. \end{cases}$$

Then by Lemma 25

$$x_i^b = 1 \leftrightarrow x_i = b. \tag{4.2}$$

Let $a = a[1:n] \in \mathbb{B}^n$ and let $x = x[1:n]$ be a vector of variables. We define the *monomial*

$$m(a) = \bigwedge_{i=1}^{n} x_i^{a_i}.$$

Then,

$$\begin{aligned} m(a) = 1 \quad &\leftrightarrow \quad \forall i \in [1:n] : x_i^{a_i} = 1 \quad \text{(Lemma 26)} \\ &\leftrightarrow \quad \forall i \in [1:n] : x_i = a_i \quad \text{(Equation 4.2)} \\ &\leftrightarrow \quad x = a. \end{aligned}$$

Thus, we have

$$m(a) = 1 \leftrightarrow x = a. \tag{4.3}$$

We define the *support* $S(f)$ of f as the set of arguments a, where f takes the value $f(a) = 1$:

$$S(f) = \{a \mid a \in \mathbb{B}^n \wedge f(a)\}.$$

[3]The term *switching function* comes from electrical engineering and stands for a Boolean function.

If the support is empty, then $e = 0$ computes f. Otherwise we set

$$e = \bigvee_{a \in S(f)} m(a).$$

Then

$$\begin{aligned} e = 1 &\leftrightarrow \exists a \in S(f) : m(a) = 1 \quad \text{(Lemma 27)} \\ &\leftrightarrow \exists a \in S(f) : a = x \quad \text{(Equation 4.3)} \\ &\leftrightarrow x \in S(f) \\ &\leftrightarrow f(x) = 1. \end{aligned}$$

Thus, equations $e = 1$ and $f(x) = 1$ have the same solutions. With Lemma 24 we conclude

$$e \equiv f(x).$$

□

The expression e constructed in the proof of Lemma 28 is called the *complete disjunctive normal form* of f.

Example: The complete disjunctive normal forms of the sum and carry functions c' defined in Table 4.1 are

$$c'(a,b,c) \equiv \overline{a} \wedge b \wedge c \vee a \wedge \overline{b} \wedge c \vee a \wedge b \wedge \overline{c} \vee a \wedge b \wedge c \tag{4.4}$$

$$s(a,b,c) \equiv \overline{a} \wedge \overline{b} \wedge c \vee \overline{a} \wedge b \wedge \overline{c} \vee a \wedge \overline{b} \wedge \overline{c} \vee a \wedge b \wedge c. \tag{4.5}$$

In algebra, one often omits the multiplication sign in order to simplify notation. Recall that the first binomic formula is usually written as

$$(a+b)^2 = a^2 + 2ab + b^2,$$

where $2ab$ is a shorthand for $2 \cdot a \cdot b$. In the same spirit we omit the $\wedge$-sign in the conjunction of literals. Thus the above identities can also be written as

$$\begin{aligned} c'(a,b,c) &\equiv \overline{a}bc \vee a\overline{b}c \vee ab\overline{c} \vee abc \\ s(a,b,c) &\equiv \overline{a}\overline{b}c \vee \overline{a}b\overline{c} \vee a\overline{b}\overline{c} \vee abc. \end{aligned}$$

Simplified Boolean expressions for the same functions are

$$\begin{aligned} c'(a,b,c) &\equiv ab \vee bc \vee ac \\ s(a,b,c) &\equiv a \oplus b \oplus c. \end{aligned}$$

The correctness can be checked in the usual brute force way by trying all 8 assignments of values in $\mathbb{B}^3$ to the variables of the expressions, or by applying the identities listed in Section 4.3.1.

In the remainder of this book, we return to the usual mathematical notation, using the equality sign for both identities and equations to be solved. Whether we deal with identities or whether we solve equations will (hopefully) be clear from the context. We will also sometimes use the equivalence sign for identities e.g., to avoid using brackets in case we give a definition with an equality sign in the right-hand side.

Chapter 5

Hardware

In Section 5.1 we introduce the classical model of *digital circuits* and show - by an induction on the depth of gates - that the computation of signals in this model is well defined. The simple argument hinges on the fact that depth in directed acyclic graphs is defined, which was established in subsection 3.4.2. We show how to transform Boolean expressions into circuits and obtain circuits for arbitrary switching functions with the help of the disjunctive normal form.

A few basic digital circuits are presented for later use in Section 5.2. This is basically the same collection of circuits as presented in [MP00] and [**?**].

In Section 5.3 we introduce a computational model consisting of digital circuits and 1 bit registers as presented in [MP00, MP98, **?**]. In the very simple Section 5.4 we define multiple 1 bit registers witch are always clocked together as n bit registers and allow the state of such registers R as *single* components $h.R$ of hardware configurations h.

Some constructions for control automate are presented in Section 5.5. They are taken from [MP00, **?**]. This section can be skipped at the first reading.

5.1 Gates and Circuits

In a nutshell, we can think of hardware as consisting of three kinds of components which are interconnected by wires: gates, storage elements, and drivers. Drivers are tricky, and their behavior cannot even properly explained in a purely digital circuit model; for details see [**?**]. Fortunately we can avoid them in the designs of this book. Gates are: AND-gates, OR-gates, $\oplus$-gates, and inverters. For the purpose of uniform language we sometimes call inverters also $NOT - gates$. In circuit schematics we use the symbols from Figure 5.1.

Intuitively, a circuit C consists of a finite set H of gates, a sequence of input signals $x[1:n]$, a set N of wires that connect them, as well as a sequence of output signals $y[1:t]$ chosen from all signals of circuit C (as illustrated in Figure 5.2). Special inputs 0 and 1 are always available for use in a circuit. Formally we specify a circuit C by the following components

- a sequence on inputs $C.x[1:n]$. We define the corresponding set of inputs of the circuit as

$$In(C) = \{C.x[i] : i \in [1:n]\}$$

- a set $C.H$ of gates which is disjoint from the set of inputs

$$C.H \cap In(C) = \emptyset$$

  The signals of a circuit then are its inputs and its gates. Moreover the constant signals 0 and 1 are always available. We collect the signals of circuit C in the set

$$Sig(C) = In(C) \cup C.H \cup \{0, 1\}$$

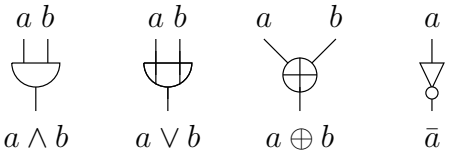


Figure 5.1: Symbols for gates in circuit schematics

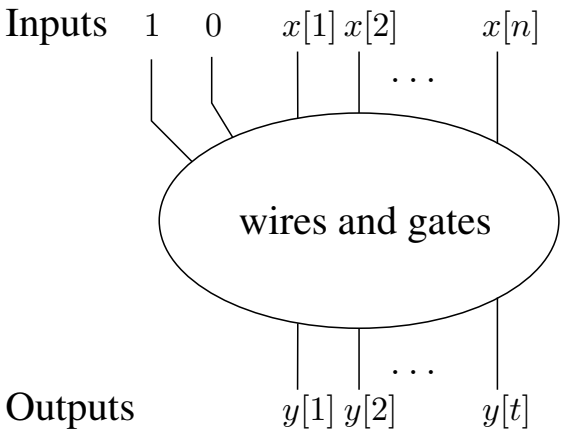


Figure 5.2: Illustration of inputs and outputs of a circuit C

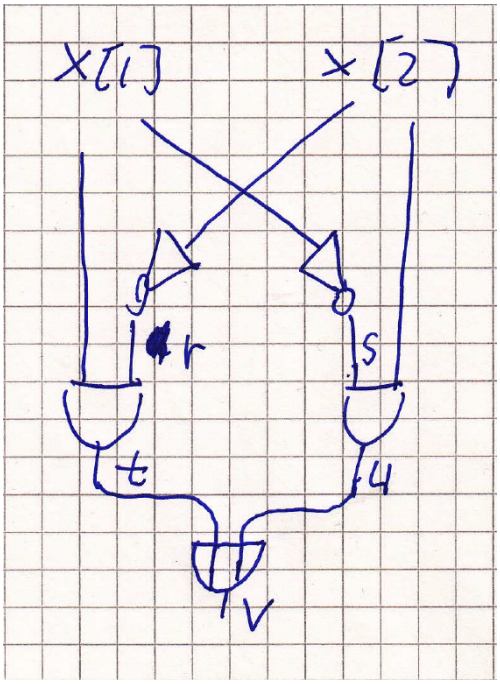


Figure 5.3: Example of a circuit

- a labelling function
$$C.\ell : H \to \{\wedge, \vee, \oplus, \neg\}$$
specifying for each gate $g \in C : H$ its type $c.\ell(g)$. Thus a gate g a $\circ$-gate if $C.\ell(g) = \circ$.

- two functions
$$C.in1, C.in2 : C.H \to Sig(C)$$
specifying for each gate $g \in C.H$ the signals, where its left input $C.in1(g)$ and its right input $C.in2(g)$ are coming from. These functions specify the wires interconnecting the gates and inputs. For inverters the second input does not matter.

- a sequence
$$C.y[1:t] \in Sig(C)^*$$
of signals of the circuit, which are taken as the outputs of the circuit.

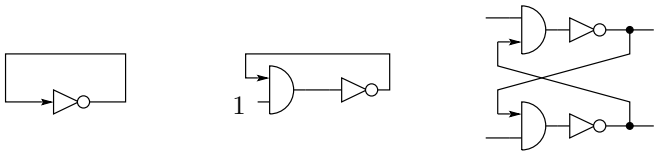


Figure 5.4: Examples for cycles in circuits

As an example we specify the circuit from figure 5.3 formally by

$$\begin{aligned} C.x[1:2] &= x[1:2] \\ C.H &= \{r, s, t, u, v\} \\ C.y[1:1] &= v \end{aligned}$$

as well as the function tables in table 5.1

g	r	s	t	u	v
$C.\ell(g)$	$\neg$	$\neg$	$\wedge$	$\wedge$	$\vee$
$C.in1(g)$	x_2	x_1	x_1	s	t
$C.in2(g)$			r	x_2	u

At first glance it is very easy to define how a circuit should work.

Definition 19 (Evaluating Signals of a Circuit)**.** *For a circuit C, we define the values $s(a)$ produced by signals $s \in Sig(C)$ for a given substitution $a = a[1:n] \in \mathbb{B}^n$ of the input signals:*

1. *if $s = x_i$ is an input:*
$$\forall i \in [1:n] : x_i(a) = a_i,$$
2. *if s is an inverter:*
$$s(a) = \overline{in1(s)(a)},$$
3. *if s is a $\circ$-gate with $\circ \in \{AND, OR, \oplus\}$:*
$$s(a) = in1(s)(a) \circ in2(s)(a).$$

Unfortunately, this is not always a definition. For three counterexamples, see Figure 5.4. Due to the cycles, one cannot find an order in which the above definition can be applied. Fortunately, defining and then forbidding cycles solves the problem. With circuits C we associate in an obvious way a graph $G(C)$ by the following rules

- the nodes of the graph are the signals of the circuit, i.e. inputs, gates ans well as the constants 0 and 1
$$G(C).V = Sig(C)$$
- we draw and edge (s, t) from signal s to signal t if t is a gate and s is an input for t
$$(s, t) \in G(C).E \leftrightarrow t \in C.H \wedge s \in \{C.in1(t), C.in2(t)$$

The sources of these graphs are the inputs as well as the constant signals 0 and 1. We restrict the definition of circuits C by requiring that their graphs $G(C)$ are cycle free. This not only excludes the counter examples above. We know from subsection 3.4.2 that in directed acyclic graph every node has a depth. Evaluating signals s in circuits in the order of their depth will always work, and we conclude by induction on the depth of gates the very reassuring

Lemma 29. *The values $s(a)$ are well defined for all signals s in a circuit C.*

Applying the definition with inputs $a[1:2] \in \mathbb{B}^2$ to the circuit of figure 5.3 gives the values in table 5.1.

$a[1:2]$	00	01	10	11
r	1	0	1	0
s	1	1	0	0
t	0	0	1	0
u	0	1	0	0
v	0	1	1	0

Obviously we can treat names g of gates as function symbols with n arguments satisfying with $x = x[1:n]$ the identities

1. if s is an inverter:
$$s(x) = \overline{in1(s)(x)},$$
2. if s is a $\circ$-gate with $\circ \in \{AND, OR, \oplus\}$:
$$s(x) = in1(s)(x) \circ in2(s)(x).$$

So we can do Boolean algebra with signal names. We can for instance summarize the last line of table 5.1 by
$$\forall a \in \mathbb{B}^2 : v(a) = 1 \leftrightarrow a_1 \oplus a_2 = 1$$
which can be written as the identity
$$v(x[1:2]) = x_1 \oplus x_2$$
or shorter
$$v = x_1 \oplus x_2$$

We can transform Boolean expressions in a trivial way into circuits.

Lemma 30. *Let e be a pure Boolean expression with variables $x[1:n]$. Then there is a circuit $C(e)$ with inputs $x[1:n]$ and a gate $g \in C : H$ such that*
$$e = g(x)$$
is an identity.

Proof. by induction on the structure of Boolean expressions. If e is a variable x_i or a constant $c \in \{0, 1\}$ we take g as the corresponding input x_i or the corresponding constant signal c in the circuit. In the induction step there are two obvious cases

- $e = \neg e'$. By induction hypothesis we have circuit $C(e')$ and gate g' such that
$$e' = g'(x)$$
We extend the circuit by an inverter g as shown in figure 5.5 a) and get
$$g(x) = \neg(g'(x)) = \neg e' = e$$
- $e = e' \circ e"$. By induction hypothesis we have circuits $C(e')$ and $C(e")$ with gates $g')$ and $g")$ satisfying
$$e' = g(x) \quad \text{and} \quad e" = g"(x)$$
We feed g' and $g"$ into a $\circ$-gate as shown in figure 5.5 b) and get
$$g(x) = g'(x) \circ g"(x) = e' \circ e" = e$$

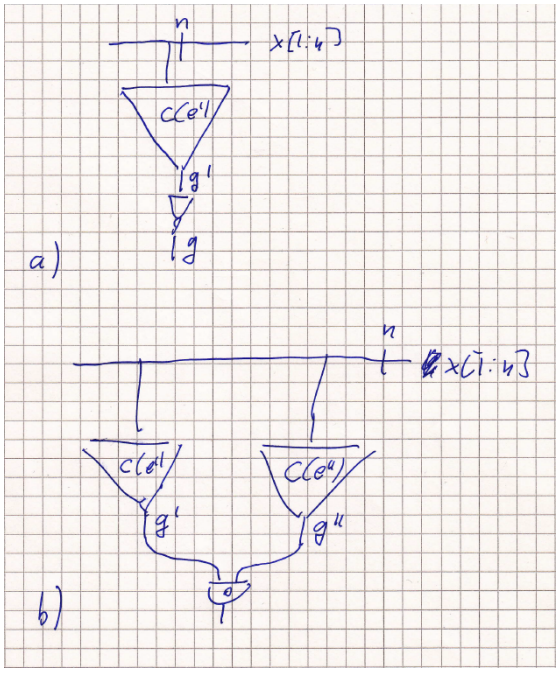


Figure 5.5: Transforming Boolean expressions to circuits. TODO: write g at output of $\circ$ gate in b)

□

Circuits for arbitrary switching functions are obtained with the help of lemma 28.

Lemma 31. *Let $f : \mathbb{B}^n \to \mathbb{B}$ be a switching function. Then there is a circuit C with inputs $x[1:n]$ and a gate g in the circuit computing f, i.e. such that*

$$g(x) = f(x)$$

is an identity

Proof. Let e be the complete disjunctive normal form for f using variables $x[1:n]$. Then

$$e = f(x)$$

By lemma 30 in circuit $C(e)$ there is a gate g with

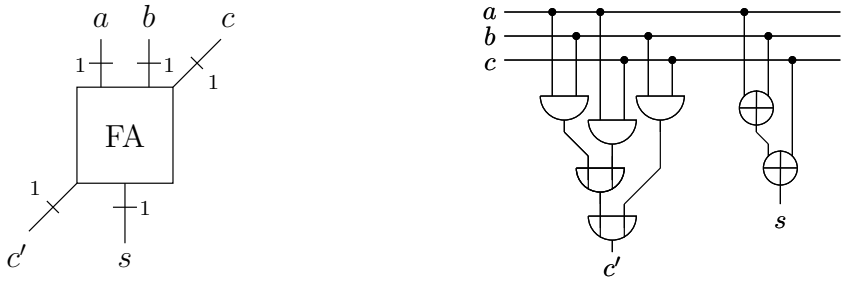
$$g(x) = e = f(x)$$

□

We conclude this section by mentioning two complexity measures for circuits C.

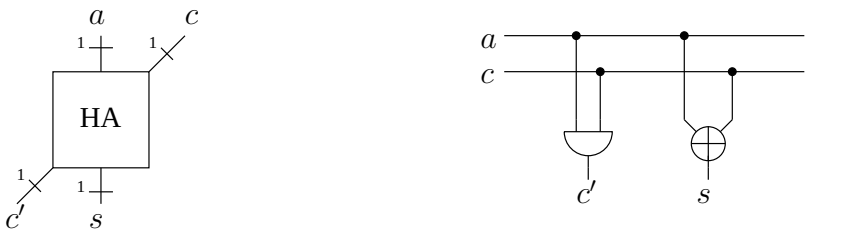
- the number of gates
$$c(C) = \#C : H$$
is an approximation for the cost of the circuit.

- the length of a longest path in the underlying graph $G(C)$ is denoted with $d(C)$ and called the depth of the circuit. It counts the maximal number of gates on a path in the circuit and is an approximation for the delay of the circuit.



(a) symbol (b) implementation

Figure 5.6: Full adder



(a) symbol (b) implementation

Figure 5.7: Half adder

5.2 Some Basic Circuits

We have shown in lemma 30 that Boolean expressions can be transformed into circuits in a very intuitive way. In Figure 5.6(b) we have transformed the simple formulas from Equation 4.4 for $c'(a,b,c)$ and $s(a,b,c)$ into a circuit. With inputs (a,b,c) and outputs (c',s) this circuit satisfies

$$\langle c', s\rangle = a + b + c.$$

A circuit satisfying this condition is called a *full adder*. We use the symbol from Figure 5.6(a) to represent this circuit in subsequent constructions. When the b-input of a full adder is known to be zero, the specification simplifies to

$$\langle c', s\rangle = a + c.$$

The resulting circuit is called a *half adder*. Symbol and implementation are shown in Figures 5.7(a) and (b). The circuit in Figure 5.8(b) is called a *multiplexer* or short *mux*. Its inputs and outputs satisfy

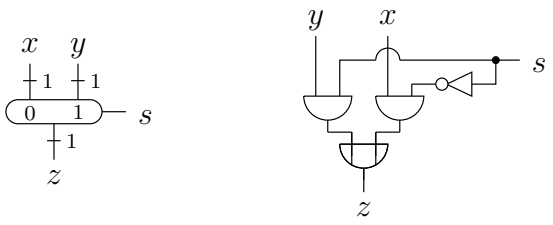
$$z = \begin{cases} x & s = 0 \\ y & s = 1. \end{cases}$$

For multiplexers we use the symbol from Figure 5.8(a). The n-bit multiplexer or short n-mux in Figure 5.9(b) consists of n multiplexers with a common select signal s. Its inputs and outputs satisfy

$$z[1:n] = \begin{cases} x[1:n] & s = 0 \\ y[1:n] & s = 1. \end{cases}$$

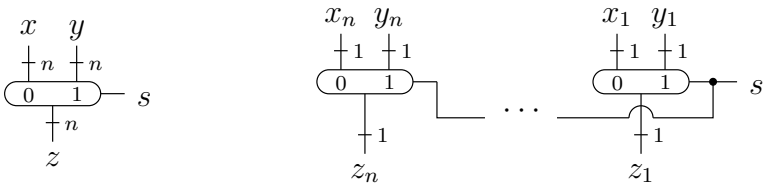
For n-muxes we use the symbol from Figure 5.9(a). Figure 5.10(a) shows the symbol for an n-bit inverter. Its inputs and outputs satisfy

$$y[1:n] = \overline{x[1:n]}.$$



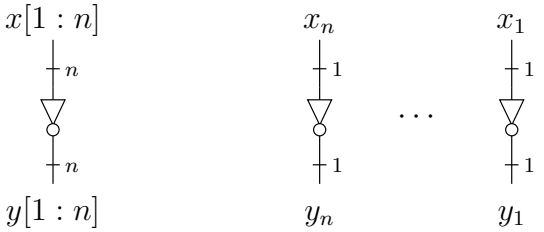
(a) symbol (b) implementation

Figure 5.8: Multiplexer



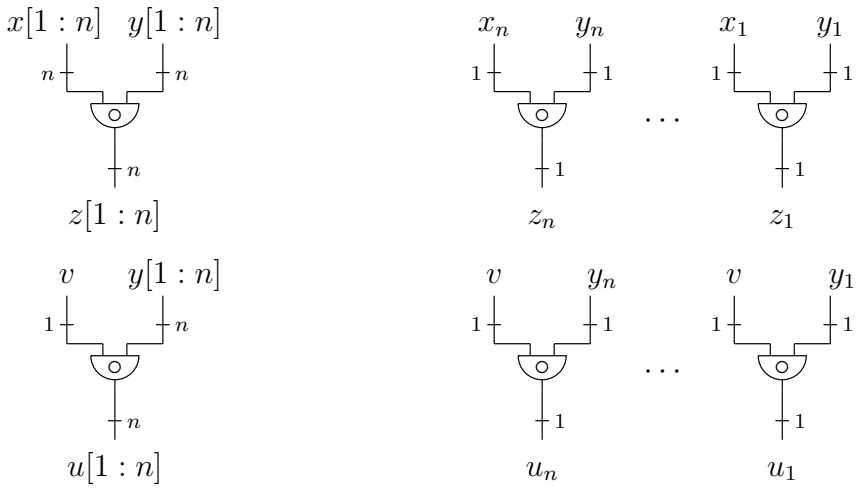
(a) symbol (b) implementation

Figure 5.9: n-bit multiplexer



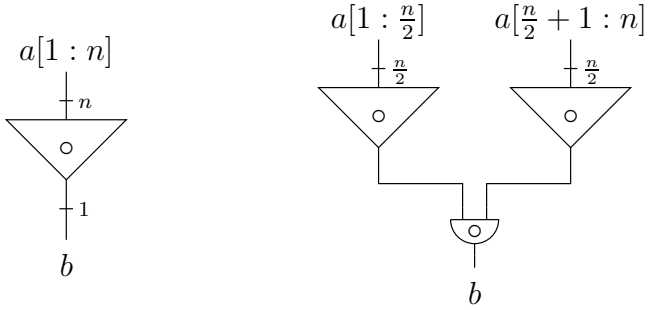
(a) symbol (b) implementation

Figure 5.10: n-bit inverter



(a) symbol (b) implementation

Figure 5.11: Gates for n-bit wide inputs



(a) symbol (b) implementation

Figure 5.12: n-bit $\circ$-tree of gates for $\circ \in \{\wedge, \vee, \oplus\}$

n-bit inverters are simply realized by n separate inverters as shown in Figure 5.10(b). For $\circ \in \{\wedge, \vee, \oplus\}$, Figure 5.11(a) shows symbols for n-bit $\circ$-gates. Their inputs and outputs satisfy

$$\begin{aligned} z[1:n] &= x[1:n] \circ y[1:n] \\ u[1:n] &= v \circ y[1:n]. \end{aligned}$$

n-bit $\circ$-gates are simply realized in the first case by n separate $\circ$-gates as shown in Figure 5.11(b). In the second case all left inputs of the gates are connected to the same input v. An n-bit $\circ$-tree has inputs $a[1:n]$ and a single output b satisfying

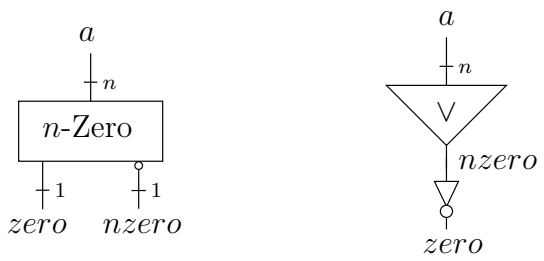
$$b = \circ_{i=1}^{n} a_i.$$

Symbol and recursive construction are shown in Figures 5.12(a) and (b). The inputs $a[1:n]$ and outputs *zero* and *nzero* of an *n-zero tester* shown in Figures 5.13(a) and (b)) satisfy

$$\begin{aligned} zero &\equiv a = 0^n \\ nzero &\equiv a \neq 0^n. \end{aligned}$$

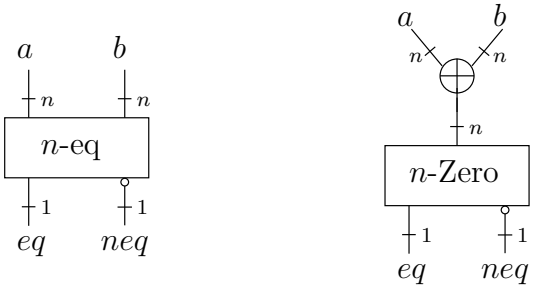
The implementation uses

$$nzero(a[1:n]) = \bigvee_{i=1}^{n} a_i \quad , \quad zero = \overline{nzero}.$$



(a) symbol (b) implementation

Figure 5.13: n-bit zero tester



(a) symbol (b) implementation

Figure 5.14: n-bit equality tester

The inputs $a[1:n]$, $b[1:n]$ and outputs eq, neq of an n-bit *equality tester* (Figures 5.14(a) and (b)) satisfy

$$\begin{aligned} eq &\equiv a = b \\ neq &\equiv a \neq b. \end{aligned}$$

The implementation uses

$$neq(a[1:n]) = nzero(a[1:n] \oplus b[1:n]) \quad , \quad eq = \overline{neq}.$$

An *n-decoder* is a circuit with inputs $x[n-1:0]$ and outputs $y[2^n-1:0]$ satisfying

$$\forall i : y_i = 1 \leftrightarrow \langle x \rangle = i.$$

A recursive construction with $k = \lceil \frac{n}{2} \rceil$ is shown in Figure 5.15. For the correctness one argues in the induction step

$$\begin{aligned} y[i \cdot 2^k + j] = 1 \quad &\leftrightarrow \quad V[i] = 1 \wedge U[j] = 1 \quad \text{(construction)} \\ &\leftrightarrow \quad \langle x[n-1:k] \rangle = i \wedge \langle x[k-1:0] \rangle = j \quad \text{(ind. hypothesis)} \\ &\leftrightarrow \quad \langle x[n-1:k] x[k-1:0] \rangle = i \cdot 2^k + j. \quad \text{(Lemma 17)} \end{aligned}$$

An *n-half decoder* is a circuit with inputs $x[n-1:0]$ and outputs $y[2^n-1:0]$ satisfying

$$y = 0^{2^n - \langle x \rangle} 1^{\langle x \rangle},$$

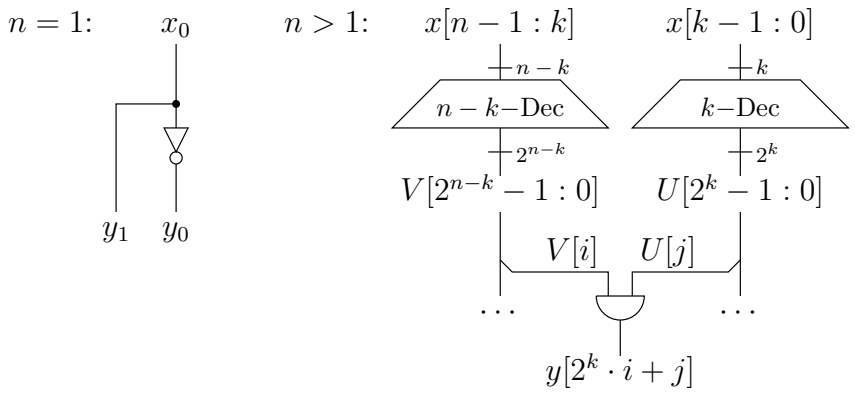


Figure 5.15: Implementation of an n-bit decoder

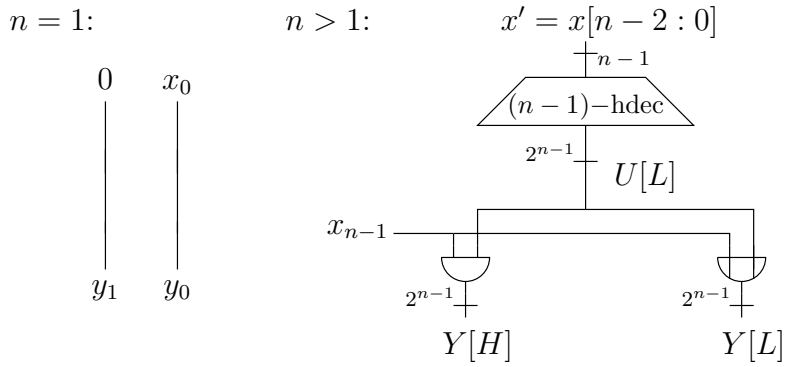


Figure 5.16: Recursive construction of n-bit half decoder

i.e. input x is interpreted as a binary number and decoded into a unary number. The remaining output bits are filled with zeros. A recursive construction of n-half decoders is shown in Figure 5.16. For the construction of n-half decoders from $(n-1)$-half decoder we divide the index range into upper and lower half:

$$L = [2^{n-1} - 1 : 0] \quad , \quad H = [2^n - 1 : 2^{n-1}].$$

Also we divide $x[n-1:0]$ into the leading bit x_{n-1} and the low order bits

$$x' = x[n-2:0].$$

Then on the induction step we conclude

$$\begin{aligned}
Y[H] \circ Y[L] &= x_{n-1} \wedge U[L] \circ (x_{n-1} \vee U(L)) \\
&= \begin{cases} 0^{2^{n-1}} \circ 0^{2^{n-1}-\langle x' \rangle} 1^{\langle x' \rangle} & : \quad x_{n-1} = 0 \\ 0^{2^{n-1}-\langle x' \rangle} 1^{\langle x' \rangle} \circ 1^{2^{n-1}} & : \quad x_{n-1} = 1 \end{cases} \\
&= \begin{cases} 0^{2^n-\langle x' \rangle} 1^{\langle x' \rangle} & : \quad x_{n-1} = 0 \\ 0^{2^n-(2^{n-1}+\langle x' \rangle)} 1^{2^{n-1}+\langle x' \rangle} & : \quad x_{n-1} = 1 \end{cases} \\
&= 0^{2^n-\langle x_{n-1}x' \rangle} 1^{\langle x_{n-1}x' \rangle} \\
&= 0^{2^n-\langle x \rangle} 1^{\langle x \rangle}.
\end{aligned}$$

5.3 Clocked Circuits

So far we have not treated storage elements yet. Now we introduce the simplest possible storage elements, namely registers capable of storing a single bit and construct a computational model

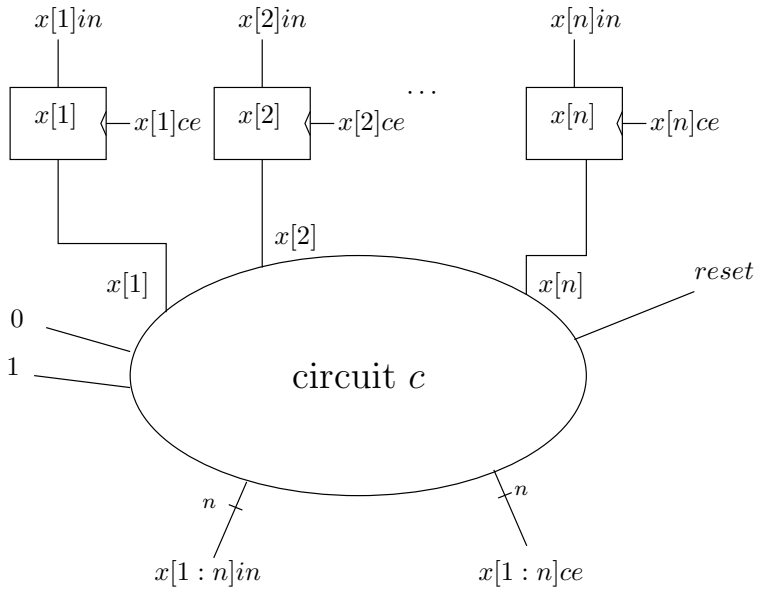


Figure 5.17: A digital clocked circuit: Every output signal $x[i]in$ of circuit c is the data input of the corresponding register $x[i]$ and every output $x[i]ce$ produced by circuit c is the clock enable input of the corresponding register

involving both circuits and these registers. All hardware constructions in this book including the construction of an entire processor with operating system support will be done in this model. Of course, we will construct more comfortable storage elements like n-bit registers and various flavors of random access memories on the way

A *digital clocked circuit* or short *clocked circuit*, as illustrated in Figure 5.17, has four components:

- a special $reset$ input,
- special inputs 0 and 1,
- a sequence $x[1:n]$ of 1 bit registers, and
- a circuit with inputs $x[1:n]$, $reset$, 0, and 1 and outputs $x[1:n]in$ and $x[1:n]ce$.

Each register $x[i]$ has

- a data input $x[i]in$,
- a clock enable input $x[i]ce$, and
- a register value $x[i]$ which is also the output signal of the register.

In the digital model we assume that register values as well as all other signals always are in $\mathbb{B}$.

A *hardware configuration* h of a clocked circuit is a snapshot of the current values of the registers:

$$h = x[1:n] \in \mathbb{B}^n.$$

A *hardware computation* is a sequence of hardware configurations where the next configuration h' is computed from the current configuration h and the current value of the $reset$ signal by a *next hardware configuration* function δ_H:

$$h' = \delta_H(h, reset).$$

In a hardware computation, we count *cycles* (steps of the digital model) using natural numbers $t \in \mathbb{N} \cup \{-1\}$. The hardware configuration in cycle t of a hardware computation is denoted by $h^t = x^t[1:n]$ and the value of signal y during cycle t is denoted by y^t.

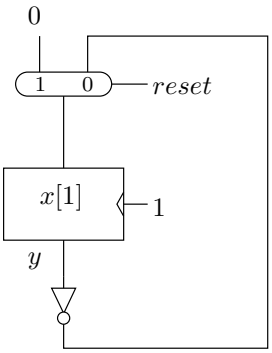


Figure 5.18: An example clocked circuit with a single register

The values of the *reset* signal are fixed. Reset is on in cycle -1 and off ever after:

$$reset^t = \begin{cases} 1 & t = -1 \\ 0 & t \geq 0. \end{cases}$$

At power up, register values are binary but unknown. We denote this sequence of unknown binary values at startup by $a[1:n]$:

$$x^{-1}[1:n] = a[1:n] \in \mathbb{B}^n.$$

The current value of a circuit signal y in cycle t is defined according to the previously introduced circuit semantics:

$$y^t = \begin{cases} \overline{in1(y)^t} & y \text{ is an inverter} \\ in1(y)^t \circ in2(y)^t & y \text{ is a } \circ\text{-gate.} \end{cases}$$

Let $x[1:n]in^t$ and $x[1:n]ce^t$ be the register input and clock enable signals computed from the current configuration $x^t[1:n]$ and the current value of the reset signal $reset^t$. Then the register value $x^{t+1}[i]$ of the next hardware configuration $x^{t+1}[1:n] = \delta_H(x^t[1:n], reset^t)$ is defined as

$$x^{t+1}[i] = \begin{cases} x[i]in^t & : \quad x[i]ce^t = 1 \\ x^t[i] & : \quad x[i]ce^t = 0 \end{cases}$$

i.e., when the clock enable signal of register $x[i]$ is active in cycle t, the register value of $x[i]$ in cycle $t+1$ is the value of the data input signal in cycle t; otherwise, the register value does not change.

Example: Consider the digital clocked circuit of Figure 5.18. There is only one register, thus we abbreviate $x = x[1]$. For cycle -1 we have

$$\begin{aligned} x^{-1} &= a[0] \\ reset^{-1} &= 1 \\ xce^{-1} &= 1 \\ xin^{-1} &= 0. \end{aligned}$$

Hence, $x^0 = 0$. For cycles $t \geq 0$ we have

$$\begin{aligned} reset^t &= 0 \\ xce^t &= 1 \\ xin^t &= y^t = \overline{x^t}. \end{aligned}$$

Hence, we get $x^{t+1} = \overline{x^t}$. An easy induction on t shows that

$$\forall t \geq 0 : x^t = (t \bmod 2).$$

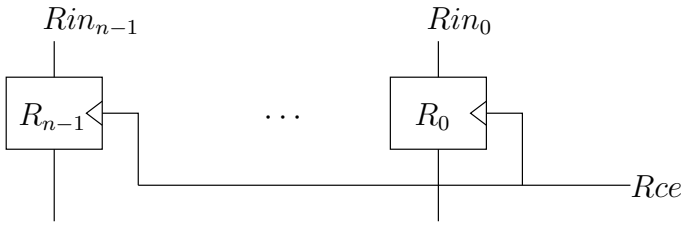


Figure 5.19: An n-bit register

5.4 Registers

Although all memory components can be built from 1 bit registers, it is inconvenient to refer to all memory bits in a computer by numbering them with an index i of a clocked circuit input $x[i]$. It is more convenient to deal with hardware configurations h and to gather groups of such bits into certain memory components $h.M$. For M we introduce n-bit registers $h.R$. In Chapter 6 we add to this no less than 5 (five) random access memory (RAM) designs.[1] As before, in a hardware computation with memory components, we have

$$h^{t+1} = \delta_H(h^t, reset^t).$$

An n-bit register R consists simply of n many 1 bit registers $R[i]$ with a common clock enable signal Rce as shown in Figure 5.19.

Register configurations are n-tuples:

$$h.R \in \mathbb{B}^n.$$

Given inputs signals $Rin(h^t)$ and $Rce(h^t)$, we obtain from the semantics of the basic clocked circuit model:

$$h^{t+1}.R = \begin{cases} Rin(h^t) & Rce(h^t) = 1 \\ h^t.R & Rce(h^t) = 0. \end{cases}$$

Recall that, from the initialization rules for 1-bit registers, after power up register content is binary but unknown (metastability is extremely rare):

$$h^0.R \in \mathbb{B}^n.$$

5.5 Finite State Transducers

Control automata (also called *finite state transducers*) are finite automata which produce an output in every step. Formally, a finite state transducer M is defined by a 6-tuple $(Z, z_0, I, O, \delta_A, \eta)$, where Z is a finite set of states, $I \subseteq \mathbb{B}^\sigma$ is a finite set of *input symbols*, $z_0 \in Z$ is called the *initial state*, $O \subseteq \mathbb{B}^\gamma$ is a finite set of *output symbols*,

$$\delta_A : Z \times I \to Z$$

is the *transition function*, and

$$\eta : Z \times I \to O$$

is the *output function*.

Such an automaton performs steps according to the following rules:

- the automaton is started in state z_0,

[1] In the more advanced text KP13] even more memory designs are used.

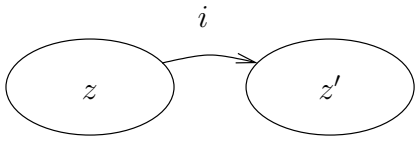


Figure 5.20: Graphical representation of a transition $z' = \delta_A(z, i)$

- if the automaton is in state z and reads input symbol in, then it outputs symbol $\eta(z, in)$ and goes to state $\delta_A(z, in)$.

If the output function does not depend on the input, i.e., if it can be written as

$$\eta : Z \to O,$$

the automaton is called a *Moore automaton*. Otherwise, it is called a *Mealy automaton*.

Automata are often visualized in graphical form. We will do this too in Section **??** when we construct several automata for the control of a cache coherence protocol. State z is drawn as a circle with a z written inside. A state transition

$$z' = \delta_A(z, i)$$

is visualized by an arrow from state z to state z' with label i as shown in Figure 5.20. Initial states are sometimes drawn as a double circle.

In what follows, we show how to implement control automata. We start with the simpler Moore automata and then generalize the construction to Mealy automata.

5.5.1 Realization of Moore Automata

Let $k = \#Z$ be the number of states of the automaton. Then, states can be numbered from 0 to $k-1$, and we can rename the states with numbers from 0 to $k-1$, taking 0 as the initial state:

$$Z = \{0, \ldots, k-1\} \quad , \quad z_0 = 0.$$

We code the current state z in a register $S \in \mathbb{B}^k$ by simple unary coding:

$$S = code(z) \leftrightarrow \forall i : S[i] \;\; = \;\; \begin{cases} 1 & z = i \\ 0 & \text{otherwise.} \end{cases}$$

A completely straightforward and naive implementation is shown in Figure 5.21. By the construction of the reset logic, we get

$$h^0.S = code(0).$$

Circuits *out* (like output) and *nexts* are constructed such that the automaton is simulated in the following sense: if $h.S = z$, i.e. state z is encoded by the hardware, then

1. $out(h) = \eta(z)$, i.e. automaton and hardware produce the same output,
2. $nexts(h) = \delta_A(z, in(h))$, i.e. in the next cycle the hardware $h'.S$ encodes the next state $\delta_A(z, in(h))$.

The following lemma states correctness of the construction shown in Figure 5.21.

Lemma 32. *Let*

$$h.S = code(z) \wedge \delta_A(z, in(h)) = z'.$$

Then,

$$out(h) = \eta(z) \wedge h'.S = code(z').$$

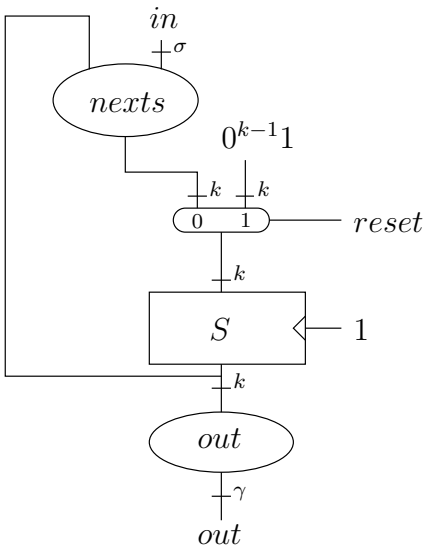


Figure 5.21: Naive implementation of a Moore automaton

For all $i \in [0 : \gamma - 1]$, we construct the i'th output simply by OR-ing together all bits $S[x]$ where $\eta(x)[i] = 1$, i.e. such that the i'th output is on in state x of the automaton:

$$out(h)[i] = \bigvee_{\eta(x)[i]=1} h.S[x].$$

A straightforward argument shows the first claim of the lemma. Assume $h.S = z$. Then,

$$h.S[x] = 1 \leftrightarrow x = z.$$

Hence,

$$\begin{aligned} & out(h)[i] = 1 \\ \leftrightarrow \quad & \bigvee_{\eta(x)[i]=1} h.S[x] = 1 \\ \leftrightarrow \quad & \exists x : \eta(x)[i] = 1 \wedge h.S[x] = 1 \\ \leftrightarrow \quad & \eta(z)[i] = 1. \end{aligned}$$

Lemma 24 gives

$$out(h)[i] = \eta(z)[i].$$

For states i, j we define auxiliary switching functions

$$\delta_{i,j} : \mathbb{B}^\sigma \to \mathbb{B}$$

from the transition function δ_A of the automaton by

$$\delta_{i,j}(in) = 1 \leftrightarrow \delta_A(i, in) = j,$$

i.e. function $\delta_{i,j}(in)$ is on if input in takes the automaton from state i to state j. Boolean formulas for functions $\delta_{i,j}$ can be constructed by Lemma 28. For each state j, component $nexts[j]$ which models the next state function is turned on in states x which transition under input in to state j according to the automaton's transition function:

$$nexts(h)[j] = \bigvee_x h.S[x] \wedge \delta_{x,j}(in(h)).$$

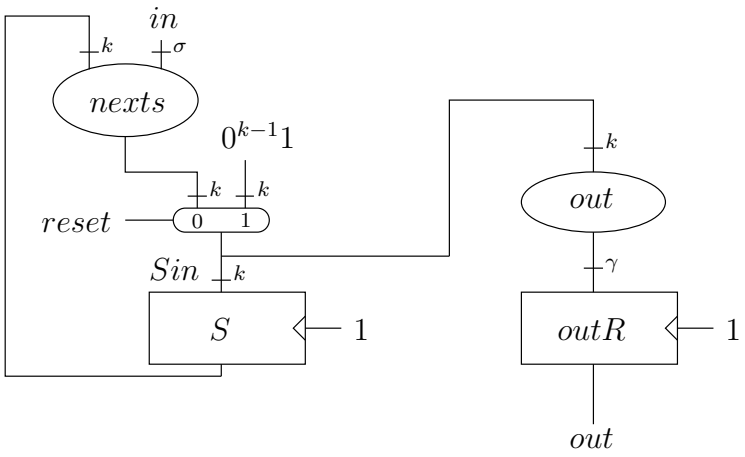


Figure 5.22: Implementation of a Moore automaton with precomputed outputs

For the second claim of the lemma let

$$\begin{aligned} h.S &= code(z) \\ \delta_A(z, in(h)) &= z'. \end{aligned}$$

For any next state j we then have

$$\begin{aligned} & nexts(h)[j] = 1 \\ \leftrightarrow \quad & \bigvee_x h.S[x] \wedge \delta_{x,j}(in(h)) = 1 \\ \leftrightarrow \quad & \delta_{z,j}(in(h)) = j \\ \leftrightarrow \quad & \delta_A(z, in(h)) = j. \end{aligned}$$

Hence,

$$nexts(h)[j] = \begin{cases} 1 & j = z' \\ 0 & \text{otherwise.} \end{cases}$$

Thus,

$$\begin{aligned} code(z') &= nexts(h) \\ &= h'.S. \end{aligned}$$

5.5.2 Precomputing Outputs of Moore Automata

The previous construction has the disadvantage that the propagation delay of circuit *out* tends to contribute to the cycle time of the circuitry controlled by the automaton. This can by avoided by precomputing the output signals of a Moore automaton as a function of the next state signals as shown in Figure 5.22.

As above, one shows

$$Sin(h) = code(z) \rightarrow out(h) = \eta(z).$$

For $h = h^{-1}$ the reset signal is active and we have

$$Sin(h^{-1}) = 0^{k-1}1 = code(0) \wedge out(h^{-1}) = \eta(0).$$

Thus,

$$h^0.S = code(0) \wedge h^0.outR = \eta(0).$$

The following lemma states correctness of the construction shown in Figure 5.22.

Lemma 33. *For $h = h^t, t \geq 0$ let*

$$h.S = code(z) \wedge \delta_A(z, in(h)) = z'.$$

Then,

$$h'.S = code(z') \wedge h'.outR = \eta(z').$$

We have $reset(h) = 0$ and hence $Sin(h) = nexts(h)$. From above we have

$$h'.S = nexts(h) = code(z')$$

and

$$h'.outR = out(h) = \eta(z').$$

Chapter 6

Five Shades of RAM

Memory components play an important role in the construction of a machine. We start in Section 6.1 with a basic construction of (static) random access memory (RAM). Next, we derive four specialized designs: read only memory (ROM) in Section 6.2, combined ROM and RAM in Section 6.3, general purpose register RAM (GPR RAM) in Section 6.4 and special purpose register RAM (SPR RAM) in Section 6.5.

For the correctness proof of a RAM construction, we consider a hardware configuration h which has the abstract state of the RAM $h.S$ as well as the hardware components implementing this RAM. The abstract state of the RAM is coupled with the state of its implementation by means of an *abstraction relation*. Given that both the abstract RAM specification and RAM implementation have the same inputs, we show that their outputs are also always the same.

The material in this section builds clearly on [MP00]. The new variations of RAMs (like general 2 port RAM), that we have introduced, are needed in later chapters. Correctness proofs for the various flavors of RAM are quite similar. Thus, if one lectures about this material, it suffices to present only a few of them in the classroom.

6.1 Basic Random Access Memory

As illustrated in Figure 6.1, an (n, a)-static RAM S or SRAM is a portion of a clocked circuit with the following inputs and outputs:

- an n-bit data input Sin,
- an a-bit address inputs Sa,
- a write signal Sw, and
- an n-bit data output $Sout$.

Internally, the static RAM contains 2^a many n-bit registers $S(x) \in \mathbb{B}^n$. Thus, it is modeled as a function

$$h.S : \mathbb{B}^a \to \mathbb{B}^n.$$

The initial content of the RAM after reset is unknown:

$$\forall x : h^0.S(x) \in \mathbb{B}^n.$$

The output of the RAM is the register content selected by the address input:

$$Sout(h) = h.S(Sa(h)).$$

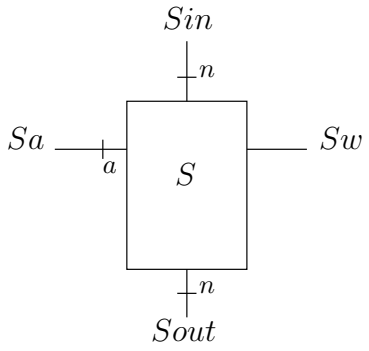


Figure 6.1: Symbol for an (n, a)-SRAM.

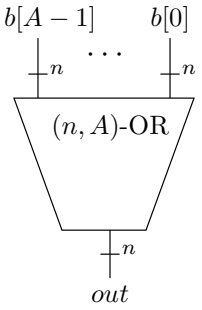


Figure 6.2: Symbol for an (n, A)-or tree.

For addresses $x \in \mathbb{B}^a$ we define the next state transition function for SRAM as

$$h'.S(x) = \begin{cases} Sin(h) & Sa(h) = x \wedge Sw(h) = 1 \\ h.S(x) & \text{otherwise.} \end{cases}$$

In order to give an implementation, we first define (n, A)-or trees. As shown in Figure 6.2, such a tree has A many input vectors $b[i] \in \mathbb{B}^n$ with $i \in [A-1:0]$, where $b[i][j]$ with $j \in [n-1:0]$ is the j-th bit of input vector $b[i]$. The outputs of the circuit $out[n-1:0]$ satisfy

$$out[j] = \bigvee_{i=0}^{A-1} b[i][j].$$

The implementation of (n, A)-or trees, for the special case where A is a power of two, is shown in Figure 6.3.

The implementation of an SRAM is shown in Figure 6.4. We use 2^a many n-bit registers $R^{(i)}$ with $i \in [0 : 2^a - 1]$ and an a-decoder with outputs $X[2^a - 1 : 0]$ satisfying

$$X(i) = 1 \leftrightarrow i = \langle Sa(h) \rangle.$$

The inputs of register $R^{(i)}$ are defined as

$$\begin{aligned} h.R^{(i)}in &= Sin(h) \\ h.R^{(i)}ce &= Sw(h) \wedge X[i]. \end{aligned}$$

For the next state computation we get

$$h'.R^{(i)} = \begin{cases} Sin(h) & i = \langle Sa(h) \rangle \\ h.R^{(i)} & \text{otherwise.} \end{cases}$$

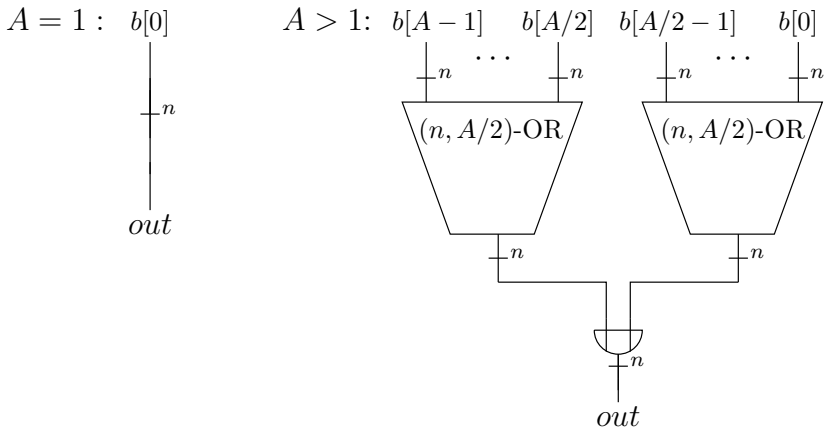


Figure 6.3: Recursive construction of an (n, A)-or tree.

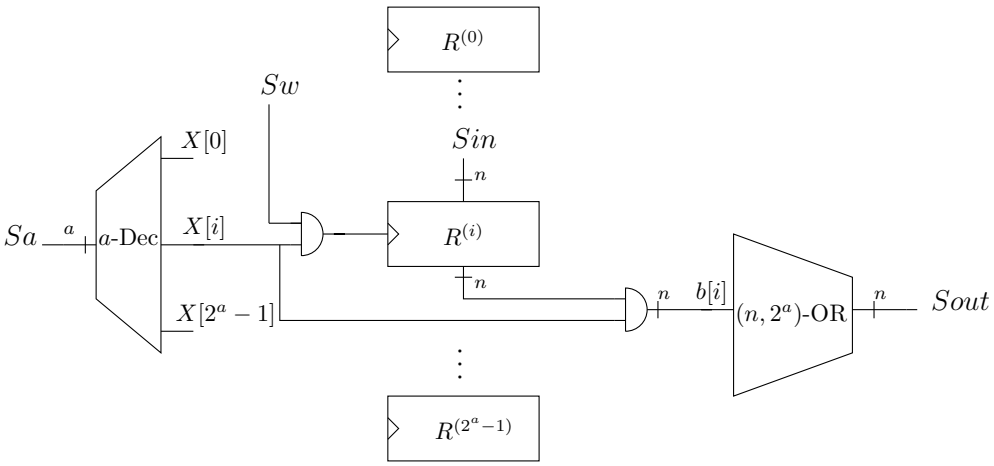


Figure 6.4: Construction of an (n, a)-SRAM.

The ith input vector $b[i]$ to the or tree is constructed as

$$\begin{aligned} b[i] &= X[i] \wedge h.R^{(i)} \\ &= \begin{cases} h.R^{(i)} & i = \langle Sa(h) \rangle \\ 0^n & \text{otherwise.} \end{cases} \end{aligned}$$

Thus,

$$\begin{aligned} Sout(h) &= \bigvee_{i=0}^{2^a-1} b[i] \\ &= h.R^{(\langle Sa(h) \rangle)}. \end{aligned}$$

As a result, when we choose

$$h.S(x) = h.R^{(\langle x \rangle)}$$

as the defining equation of our abstraction relation, the presented construction implements an SRAM.

6.2 Read Only Memory (ROM)

An (n, a)-ROM is is a memory with a drawback and an advantage. The drawback: it can only be read. The advantage: its content is known after power up. It is modeled by a mapping $S : \mathbb{B}^a \to \mathbb{B}^n$,

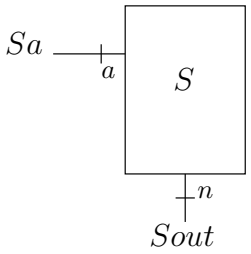


Figure 6.5: Symbol of an (n, a)-ROM.

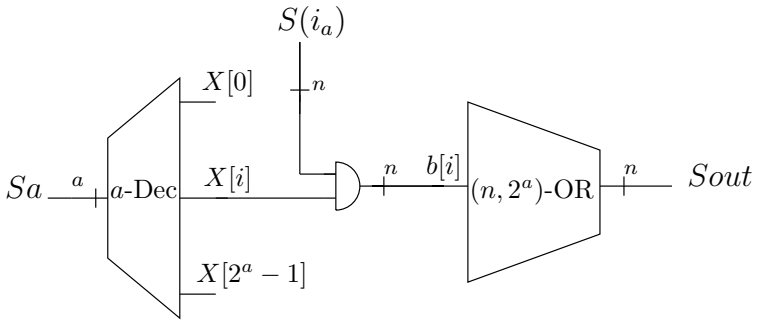


Figure 6.6: Construction of an (n, a)-ROM.

which does not depend on the hardware configuration h. The construction is obtained by a trivial variation of the basic RAM design from Figure 6.4: replace each register $R^{(i)}$ by the constant input $S(bin_a(i)) \in \mathbb{B}^n$. Since the ROM cannot be written, there are no data in, write, or clock enable signals; the hardware constructed in this way is a circuit. Symbol and construction are given in Figures 6.5 and 6.6.

6.3 Combining RAM and ROM

It is often desirable to implement some small portion of memory by ROM and the remaining large part as RAM. The standard use for this is to store boot code in the ROM. Since, after power up, the memory content of RAM is unknown, computation will not start in a meaningful way unless at least *some* portion of memory contains code that is known after power up. The reset mechanism of the hardware ensures that processors start by executing the program stored in the ROM. This code usually contains a so called *boot loader* which accesses a large and slow memory device – like a disk – to load further programs, e.g. an operating system, to be executed from the device.

For $r < a$ we define a combined (n, r, a)-RAM-ROM S as a device that behaves for small addresses $a = 0^{a-r}b$ with $b \in \mathbb{B}^r$ like ROM and on the other addresses like RAM. Just like an ordinary (n, a)-RAM, we model the state of the (n, r, a)-RAM-ROM as

$$h.S : \mathbb{B}^a \to \mathbb{B}^n$$

and define its output as

$$Sout(h) = h.S(Sa(h)).$$

Write operations, however, affect only addresses larger than $0^{a-r}1^r$:

$$h'.S(x) = \begin{cases} h^0.S(x) & x[a-1:r] = 0^{a-r} \\ Sin(h) & x = Sa(h) \wedge Sw(h) \wedge x[a-1:r] \neq 0^{a-r} \\ h.S(h) & \text{otherwise.} \end{cases}$$

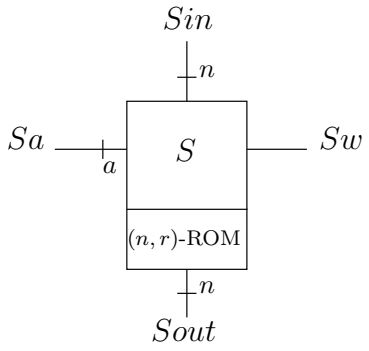


Figure 6.7: Symbol of an (n, r, a)-RAM-ROM.

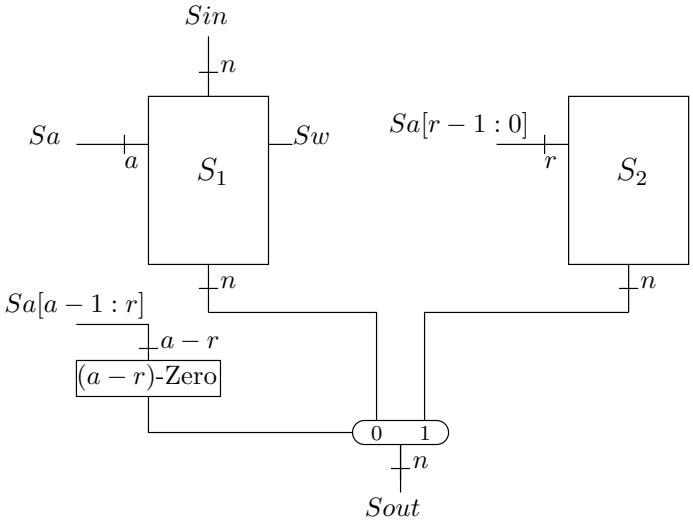


Figure 6.8: Construction of an (n, r, a)-RAM-ROM.

The symbol for an (n, r, a)-RAM-ROM and a straight forward implementation involving an (n, a)-SRAM, an (n, r)-ROM, and an $(a - r)$-zero tester is shown in Figures 6.7 and 6.8.

6.4 Three Port RAM for General Purpose Registers

An (n, a)-GPR-RAM is a three port RAM that we use later for general purpose registers. As shown in Figure 6.9, it has the following inputs and outputs:

- an n-bit data input Sin,
- three a-bit address inputs Sa, Sb, Sc,
- a write signal Sw, and
- two n-bit data outputs $Souta, Soutb$.

As for ordinary SRAM, the state of the 3-port RAM is a mapping

$$h.S : \mathbb{B}^a \to \mathbb{B}^n.$$

Reads are controlled by address inputs $Sa(h)$ and $Sb(h)$:

$$\begin{aligned} Souta(h) &= h.S(Sa(h)) \\ Soutb(h) &= h.S(Sb(h)). \end{aligned}$$

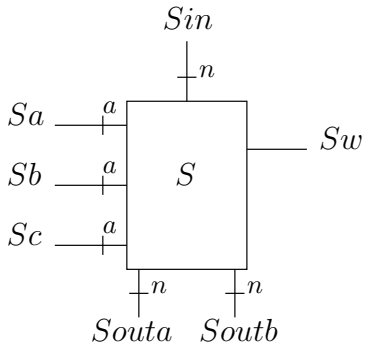


Figure 6.9: Symbol of an (n, a)-GPR RAM.

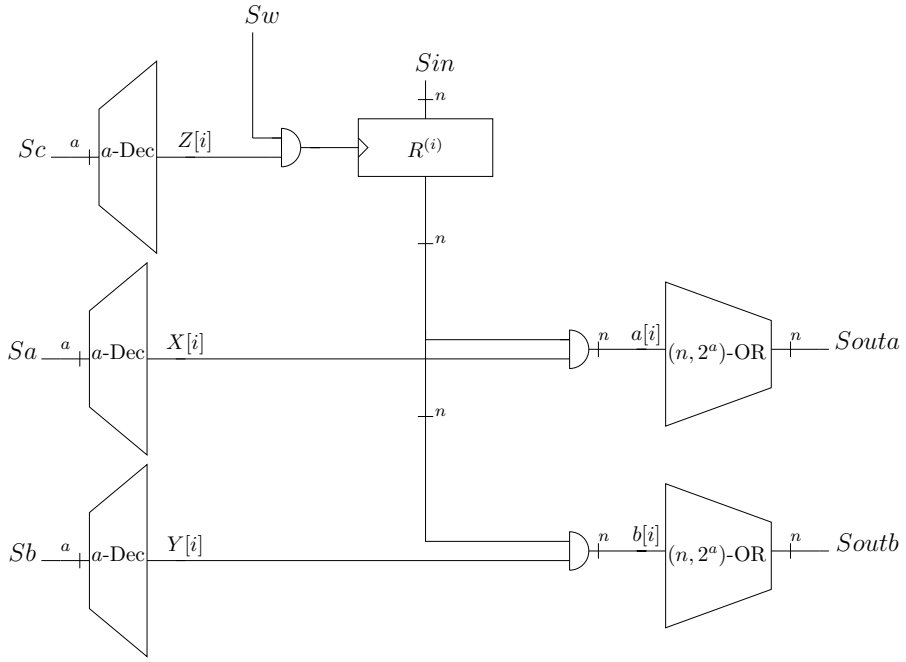


Figure 6.10: Construction of an (n, a)-GPR-RAM.

Writing is performed under control of address input $Sc(h)$:

$$h'.S(x) = \begin{cases} Sin(h) & Sc(h) = x \wedge Sw(h) = 1 \\ h.S(x) & \text{otherwise.} \end{cases}$$

The implementation shown in Figure 6.10 is a straightforward variation of of the design for ordinary SRAM. One uses three different a-decoders with outputs $X[0 : 2^a - 1]$,$Y[0 : 2^a - 1]$, $Z[0 : 2^a - 1]$ satisfying

$$\begin{aligned} X[i] = 1 &\quad \leftrightarrow i = \langle Sa(h) \rangle \\ Y[i] = 1 &\quad \leftrightarrow i = \langle Sb(h) \rangle \\ Z[i] = 1 &\quad \leftrightarrow i = \langle Sc(h) \rangle. \end{aligned}$$

Clock enable signals are derived from the decoded Sc address:

$$R^{(i)}ce = Z[i] \wedge Sw(h).$$

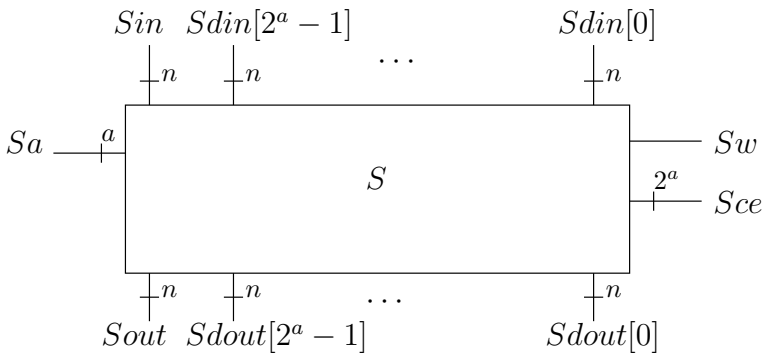


Figure 6.11: Symbol of an (n, a)-SPR RAM.

Outputs $Souta, Soutb$ are generated by two $(n, 2^a)$-or trees with inputs $a[i]$, $b[i]$ satisfying

$$\begin{aligned} a[i] &= X[i] \wedge h.R^{(i)} \\ Souta(h) &= \bigvee a[i] \\ b[i] &= Y[i] \wedge h.R^{(i)} \\ Soutb(h) &= \bigvee b[i]. \end{aligned}$$

6.5 SPR RAM

An (n, a)-SPR-RAM as shown in Figure 6.11 is used for the realization of special purpose register files. It behaves both as an (n, a)-RAM and as a set of 2^a many n bit registers. It has the following inputs and outputs:

- an n-bit data input Sin,
- an a-bit address input Sa,
- an n-bit data output $Sout$,
- a write signal Sw,
- for each $i \in [0 : 2^a - 1]$ an individual n-bit data input $Sdin[i]$ for register $R^{(i)}$,
- for each $i \in [0 : 2^a - 1]$ an individual n-bit data output $Sdout[i]$ for register $R^{(i)}$, and
- for each $i \in [0 : 2^a - 1]$ an individual clock enable signal $Sce[i]$ for register $R^{(i)}$.

Ordinary data output is generated as usual, and the individual data outputs are simply the outputs of the internal registers:

$$\begin{aligned} Sout(h) &= h.S(Sad(h)) \\ Sdout(h)[i] &= h.S(bin_a(i)). \end{aligned}$$

Register updates to $R^{(i)}$ can be performed either by Sin for regular writes or by $Sdin[i]$ if the special clock enables are activated. Special writes take precedence over ordinary writes:

$$h'.S(x) = \begin{cases} Sdin(h)[\langle x \rangle] & Sce(h)[\langle x \rangle] = 1 \\ Sin(h) & Sce(h)[\langle x \rangle] = 0 \wedge Sw(h) = 1 \\ h.S(x) & \text{otherwise.} \end{cases}$$

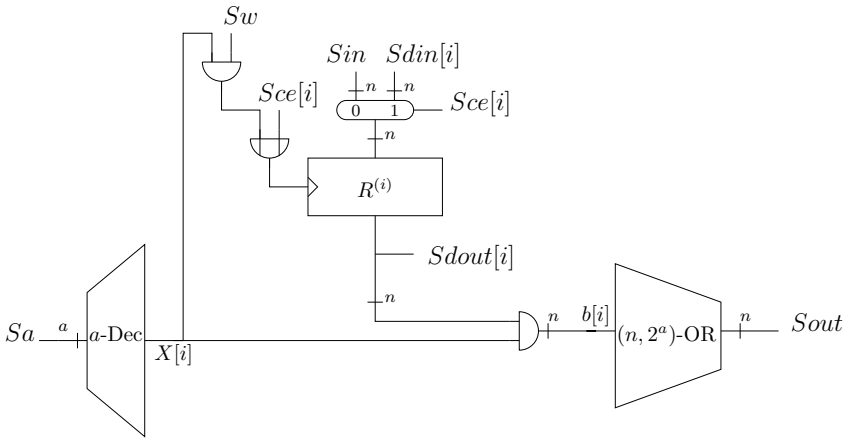


Figure 6.12: Construction of an (n, a)-SPR RAM.

A single address decoder with outputs $X[i]$ and a single or-tree suffices. Figure 6.12 shows the construction satisfying

$$\begin{aligned} R^{(i)}ce &= Sce(h)[i] \vee X[i] \wedge Sw(h) \\ R^{(i)}in &= \begin{cases} Sdin(h)[i] & Sce(h)[i] = 1 \\ Sin(h) & \text{otherwise.} \end{cases} \end{aligned}$$

Chapter 7

Arithmetic Circuits

For later use in processors with the MIPS instruction set architecture (ISA), we construct several circuits: various flavors of adders and incrementers are presented in section 7.1. In particular we construct in subsection 7.1.4 so called carry-look-ahead adders which, for arguments of length n have at the same time cost linear in n and depth logarithmic in n. The slightly advanced construction is based on the parallel prefix circuits from section 5.2. It can be skipped, but we recommend to cover this construction. It is after all the real thing and we feel everybody should be given the chance to know it.

An arithmetic unit (AU) for binary and two's complement numbers is studied in Section 7.2. In our view understanding the proofs of this section is a must for anyone wishing to understand fixed point arithmetic. With the help of the AU we construct in Section 7.3 an arithmetic logic unit (ALU) for the MIPS instruction set architecture (ISA) in a straight forward way. Differences to to [MP00] are simply due to differences in the encoding of ALU operations between the MIPS ISA considered here and the DLX ISA considered in [MP00]. Section 7.4 contains a rudimentary shifter construction supporting just logical right shifts. The short and technical section 7.5 we construct a branch condition evaluation (BCE) unit which is later used to implement the branch instructions of the MIPS ISA. In lectures this section should be covered by a reading assignment.

In textbooks on computer architecture like [MP00, **?**] the counter part of this chapter includes general shifter constructions. Shifter constructions are interesting and we do cover them in the classroom.

7.1 Adder and Incrementer

An *n-adder* is a circuit with inputs $a[n-1:0] \in \mathbb{B}^n$, $b[n-1:0] \in B^n$, $c_0 \in \mathbb{B}$ and outputs $c_n \in \mathbb{B}$ and $s[n-1:0] \in \mathbb{B}^n$ satisfying the specification

$$\langle c_n, s[n-1:0]\rangle = \langle a[n-1:0]\rangle + \langle b[n-1:0]\rangle + c_0.$$

We use the symbol from Figure 7.1 for n-adders.

A full adder is obviously a 1-adder. In the following, we present constructions for n-adders.

7.1.1 Carry-Chain Adder

A recursive construction of a very simple carry chain adder is shown in Figure 7.2. correctness follows directly from the correctness of the basic addition algorithm for binary numbers (Lemma 19).

An *n-incrementer* is a circuit with inputs $a[n-1:0] \in \mathbb{B}^n, c_0 \in \mathbb{B}$ and outputs $c_n \in \mathbb{B}$ and $s[n-1:0] \in \mathbb{B}^n$ satisfying

$$\langle c_n, s[n-1:0]\rangle = \langle a[n-1:0]\rangle + c_0.$$

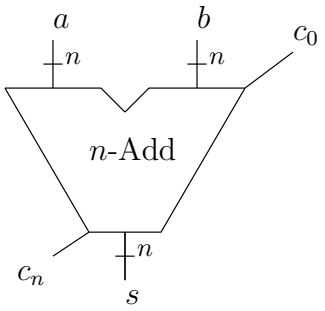


Figure 7.1: Symbol of an n-adder.

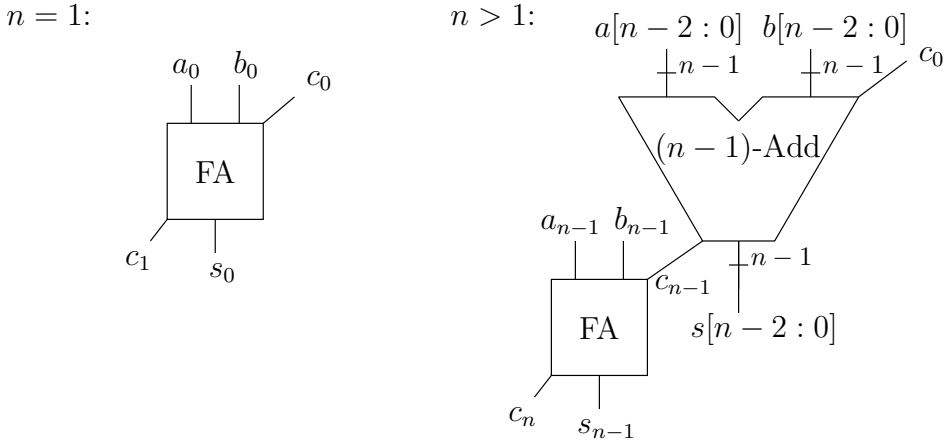


Figure 7.2: Recursive construction of a carry chain adder.

We use the symbol from Figure 7.3 for n-incrementers.

Obviously, incrementers can be constructed from n-adders by tying the b input to 0^n. As shown in Section 5.2 a full adders whose b input is tied to zero can be replaced with a half adder. This yields the construction of carry chain incrementers shown in Figure 7.4.

We introduce special symbols $+_n$ and $-_n$ to denote addition and subtraction of n bit binary numbers mod 2^n:

$$\begin{aligned} a +_n b &= bin_n(\langle a\rangle + \langle b\rangle \bmod 2^n) \\ a -_n b &= bin_n(\langle a\rangle - \langle b\rangle \bmod 2^n). \end{aligned}$$

7.1.2 Conditional-Sum Adder and Incrementer

In section 5.1 we denoted cost and delay of circuits S by $c(S)$ resp. $d(S)$. We denote the complexity of n bit carry chain adders by $c(n)$ and their depth by $d(n)$ and read off from the constructions the following so called *difference equations*:

$$\begin{aligned} c(1) &= c(FA) \\ c(n) &= c(n-1) + c(FA) \\ d(1) &= d(FA) \\ d(n) &= \leq d(n-1) + d(FA) \end{aligned}$$

A trivial induction shows

$$\begin{aligned} c(n) &= n \cdot c(FA) \\ d(n) &= d \cdot d(FA) \end{aligned}$$

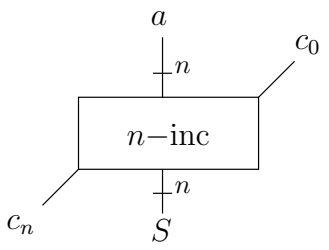


Figure 7.3: Symbol of an n-incrementer.

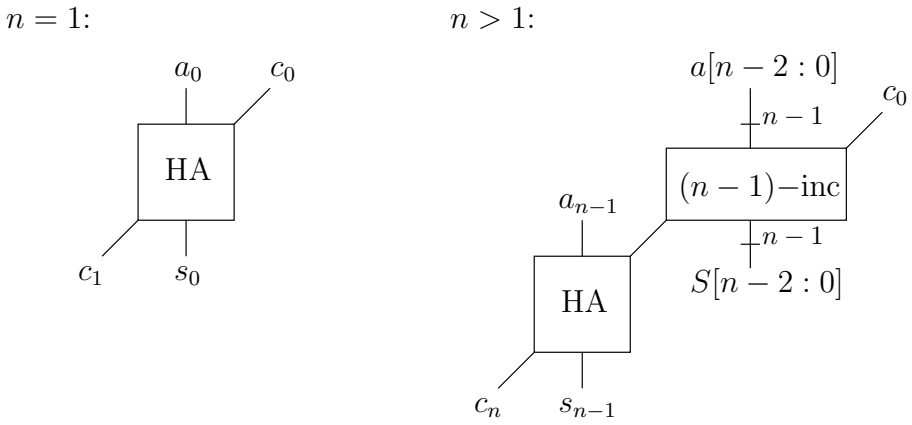


Figure 7.4: Recursive construction of a carry chain incrementer.

Replacing full adders by half adders one can make a completely analogous argument can be made for carry chain incrementers. Thus we can summarize the asymptotic complxity of theses circuits in

Lemma 34. *n bit carry chain adders and incrementers have cost $O(n)$ and delay $O(n)$*

Linear cost $O(n)$ is all one can hope for. At first sight, the delay formula also seems optimal, because by definition appears to be inherently sequential. One is tempted to argue, that one cannot compute the carry c_n at position n before one knows the carry c_{n-1} at position $n-1$. But trying to turn this argument into a proof fails because there are *much* faster adder constructions. The simplest one are so called n bit *conditional sum adders* or short n-CSAs. They are defined inductively. We take a full adder as a 1-CSA. For even n the construction of an n-CSA from 3 $n/2$-CSA is shown in figure 7.5

One splits the inputs $a, b \in \mathbb{B}^n$ into upper and lower halves:

$$\begin{array}{rclcrcl} a_H & = & a[n-1:\frac{n}{2}] & & a_L & = & a[\frac{n}{2}-1:0] \\ b_H & = & b[n-1:\frac{n}{2}] & & b_L & = & b[\frac{n}{2}-1:0] \end{array}$$

Then, one exploits the fact that the carry of the binary addition of a_L and b_L is either 0 or 1: instead of using carry $c_{\frac{n}{2}}$ as an input to the computation of the upper half of the sum, one simply computes binary representations of both $\langle a_H \rangle + \langle b_H \rangle + 0$ and $\langle a_H \rangle + \langle b_H \rangle + 1$, one of these two will be the correct result which is chose with a multiplexer controlled by signal $c_{\frac{n}{2}}$. Correctness of this construction follows from the simple lemma 20. We use now $c(n)$ and $d(n)$ to denote cost and depth of n-CSAs and obtain from the construction the difference equations.

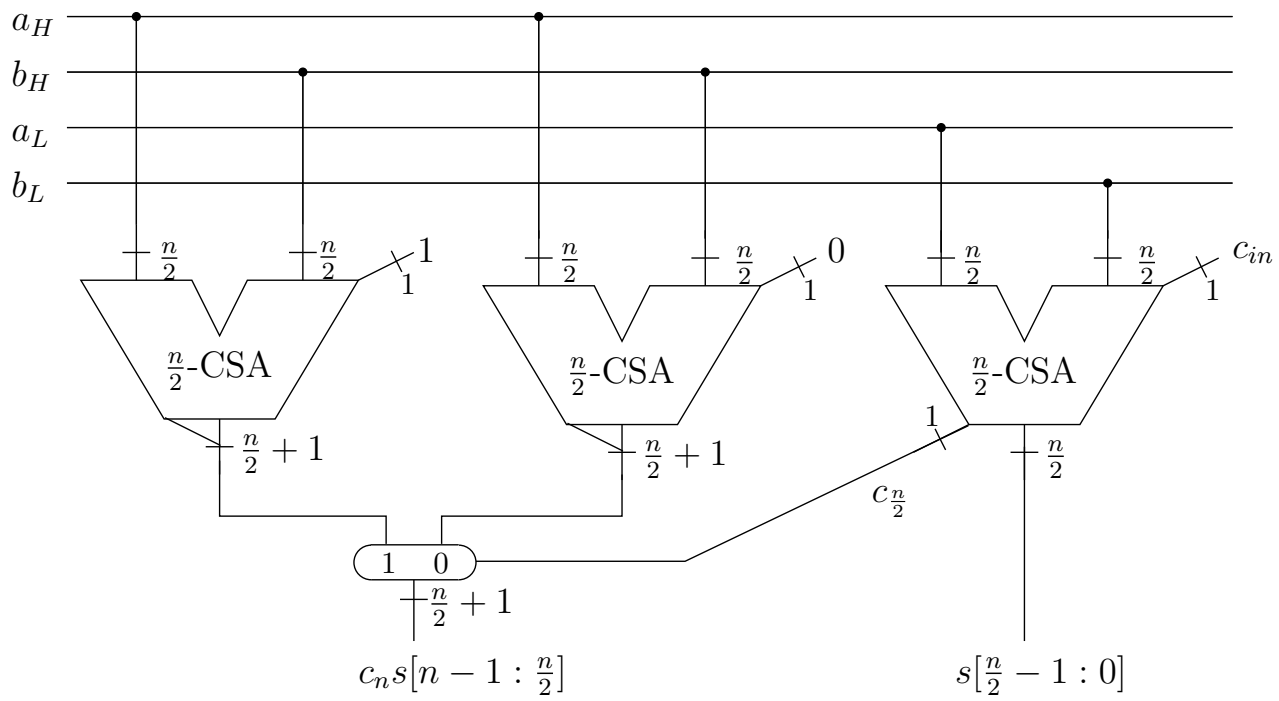


Figure 7.5: Recursive construction of a conditional sum adder.

$$
\begin{aligned}
d(1) &= d(FA) \\
d(n) &= d(n/2) + d(\text{MUX}) \\
&= d(n/2) + 3 \\
c(1) &= c(FA) \\
c(n) &= 3 \cdot c(n/2) + c(((n/2)+1)\text{-MUX}) \\
&\geq 3 \cdot c(n/2)
\end{aligned}
$$

In general, difference equations are solved in two steps: i) guessing the solution and ii) showing by induction, that the guessed solution satisfies the equations. As is usual in proofs by inductions, finding the induction hypothesis is the hard part. Fortunately there is a simple heuristics, that will work for us: we repeatedly apply the inductive definition until we see enough to guess the solution. Let

$$n = 2^k$$

be a power of two so that starting from n we can apply the recursive construction repeatedly until we arrive at $n = 1$. The heuristics gives for the depth of CSA's

$$
\begin{aligned}
d(n) &= d(n/2) + 3 \\
&= d(n/4) + 2 \cdot 3 \\
&= d(n/8) + 3 \cdot 3
\end{aligned}
$$

At this point we might guess for all x:

$$d(n) = d(n/2^x) + x \cdot 3$$

which, for

$$x = k = \log n$$

gives

$$d(n) = d(FA) + 3 \cdot \log n$$

That this guess indeed satisfies the difference equations is a trivial exercise that we omit. Unfortunately, conditional sum adders are quite expensive. Repeated application of the lower bound for their cost gives

$$\begin{aligned} c(n) &\geq 3 \cdot c(n/2) \\ &\geq 3^2 c(n/4) \ldots \\ &\geq 3^x \cdot c(n/2^x) \\ &\geq 3^{\log n} \cdot c(1) \\ &= 3^{\log n} \cdot c(FA) \end{aligned}$$

Again, proving by induction that this guess satisfies the difference equations is easy. We estimate the term 3^{logn}:

$$\begin{aligned} 3^{\log n} &= (2^{\log 3})^{\log n} \\ &= 2^{(\log 3)\cdot(\log n)} \\ &= (2^{\log n})^{\log 3} \\ &= n^{\log 3} \end{aligned}$$

We summarize the estimates in

Lemma 35. *n-CSAs have depth $O(logn)$ and cost at least $C(FA) \cdot n^{\log 3}$*

As $\log 3 = 1.73\ldots$ the cost estimate is way above the cost of carry chain adders.

7.1.3 Parallel Prefix Circuits

In life and in science sometimes extremely good things are possible. In life it is possible to be rich and healthy at the same time. In adder construction it is possible to achieve the asymptotic cost of carry chain adders and the asymptotic delay of conditional sum adders at the same time in so callled n bit *carry-look-ahead adders* or short n-CLAs. The fundamental auxiliary circuits that are used in their construction are so called *n-parallel prefix circuits*

An *n-parallel prefix circuit* for an associative function $\circ : M \times M \to M$ is a circuit with inputs $x[n-1:0] \in M^n$ and outputs $y[n-1:0] \in M^n$ satisfying

$$y_0 = x_0 \quad , \quad y_{i+1} = x_{i+1} \circ y_i.$$

We construct it in a slightly generalized circuit model where signals take values in M and computation is done by gates for function $\circ$. Cost and delay in this model are defined by counting $\circ$-gates in the circuit resp. on the longest path. We use the construction later with $M = \mathbb{B}^2$. For even n a recursive construction of an n-parallel prefix circuit based on $\circ$-gates is shown in Figure 7.6. For odd n one can realize an $(n-1)$-bit parallel prefix from Figure 7.6 and compute output y_{n-1} as

$$y_{n-1} = x_{n-1} \circ y_{n-2}$$

using one extra $\circ$-gate.

For the correctness of the construction we first observe that

$$\begin{aligned} x'_i &= x_{2i+1} \circ x_{2i} \\ y_{2i} &= x_{2i} \circ y'_{i-1} \\ y_{2i+1} &= y'_i. \end{aligned}$$

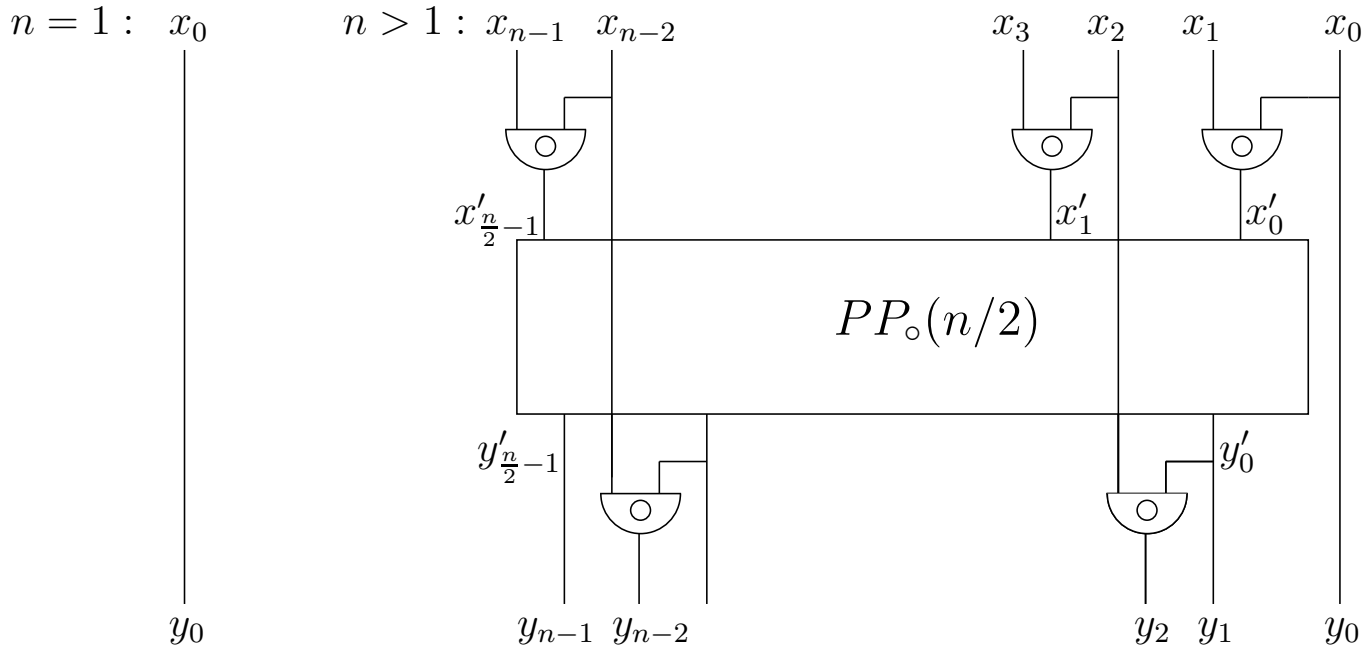


Figure 7.6: Recursive construction of an n-bit parallel prefix circuit of the function $\circ$ for an even n

We first show that odd outputs of the circuit are correct. For $i = 0$ we have

$$\begin{aligned} y_1 &= y'_0 \quad \text{(construction)} \\ &= x'_0 \quad \text{(ind. hypothesis } PP_\circ(n/2)) \\ &= x_1 \circ x_0 \quad \text{(construction).} \end{aligned}$$

For $i > 0$ we conclude

$$\begin{aligned} y_{2i+1} &= y'_i \quad \text{(construction)} \\ &= x'_i \circ y'_{i-1} \quad \text{(ind. hypothesis } PP_\circ(n/2)) \\ &= (x_{2i+1} \circ x_{2i}) \circ y'_{i-1} \quad \text{(construction)} \\ &= x_{2i+1} \circ (x_{2i} \circ y'_{i-1}) \quad \text{(associativity)} \\ &= x_{2i+1} \circ y_{2i} \quad \text{(construction).} \end{aligned}$$

For even outputs of the circuit we easily conclude

$$\begin{aligned} y_0 &= x_0 \quad \text{(construction)} \\ i > 0 \rightarrow y_{2i} &= x_{2i} \circ y'_{i-1} \quad \text{(construction)} \\ &= x_{2i} \circ y_{2i-1} \quad \text{(construction).} \end{aligned}$$

We denote by $c(n)$ and $d(n)$ cost and delay of n-parallel-prefix circuits constructed in this way. From the construction we read off the difference equations

$$\begin{aligned} d(2) &= 1 \\ d(n) &= d(n/2) + 2 \\ c(2) &= 1 \\ c(n) &\leq c(n/2) + n \end{aligned}$$

Applying the heuristics for the depth gives

$$d(n) = d(n/2^x) + 2 \cdot x$$

The recursion must be stopped when

$$\begin{aligned} n/2^x &= 2 \\ \leftrightarrow 2^x &= n/2 \\ \leftrightarrow x &= \log n - 1 \end{aligned}$$

This gives the conjecture

$$d(n) = d(2) + 2 \cdot (logn - 1) = 2 \cdot logn - 1$$

which is easily verified by induction.

For the cost we get the estimate

$$\begin{aligned} c(n) &\leq c(n/2) + n \\ &= \leq c(n/4) + n + n/2 \\ &\leq c(n/8) + n + n/2 + n/4 \ldots \\ \leq c(n/x) + n \cdot \sum_{i_0}^{x-1} (1/2)^i \\ \leq c(2) + n \cdot \sum_{i_0}^{\log n - 2} (1/2)^i \end{aligned}$$

We guess

$$c(n) \leq 2 \cdot n$$

which is verified by an easy induction proof. We summarize

Lemma 36. *There are n-parallel-prefix circuits with cost $O(n)$ and depth $O(\log n)$.*

7.1.4 Carry-Look-Ahead Adders

For $a[n-1:0], b[n-1]$ and indices $i \leq j$ we define

$$p_{i,j}(a,b) \equiv \langle a[j:i] \rangle + \langle a[j:i] \rangle = \langle 1^{j-i+1} \rangle$$

This is the case if $c_i = c_{j+1}$, i.e. if carry c_i is *propagated* by positions i to j of the operands to position $j+1$. Similarly, we define

$$g_{i,j}(a,b) \equiv \begin{cases} \langle a[j:i] \rangle + \langle a[j:i] \rangle = \langle 10^{j-i+1} \rangle & i > 0 \\ \langle a[j:i] \rangle + \langle a[j:i] \rangle = \langle 10^{j-i+1} \rangle + c_0 & i = 0 \end{cases}$$

i.e. if positions i to j of the operands *generate* a carry independent of c_i. We abbreviate with $p_{i,j}$ and $g_{i,j}$ and observe for $j = i$

$$\begin{aligned} p_{i,i} &= a_i \oplus b_i \\ g_{i,i} &= \begin{cases} a_i \wedge b_i & i > 0 \\ a_0 \wedge b_0 \vee a_0 \wedge c_0 \vee b_0 \vee c_0 & i = 0 \end{cases} \end{aligned}$$

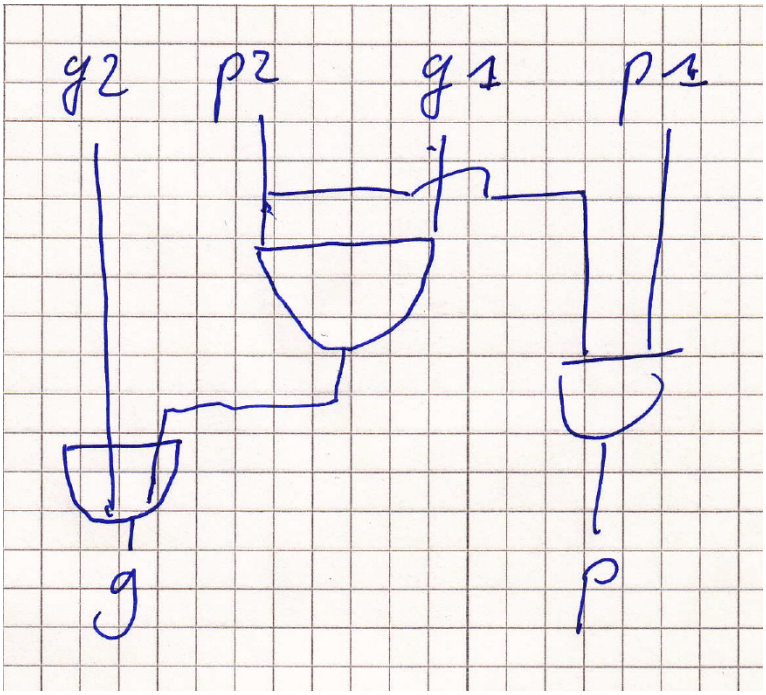


Figure 7.7: Circcuir replacing $\circ$-gates in the parallel prefix circuit

Now consider indices $i \leq k$ and $k+1 \leq j$ delimiting adjacent intervals of indices, and suppose we know already functions $g_{i,k}, p_{i,k}, g_{k+1,j}, p_{k+1,j}$ are already known. The the signals for the combined interval $i:j$ can be computed as

$$\begin{aligned} p_{i,j} &= p_{i,k} \wedge p_{k+1,j} \\ g_{i,j} &= g_{k+1,j} \vee g_{i,k} \wedge p_{k+1,j} \end{aligned}$$

This computation can be performed by the circuit in figure 7.7 which takes inputs $(g1, p1)$ and $(g2, p2)$ from $M = \mathbb{B}^2$ and produces as output in $\mathbb{B}^2$

$$\begin{aligned} (g, p) &= (g2 \vee g1 \wedge p2, p1 \wedge p2) \\ &= (g2, p2) \circ (g1, p1) \end{aligned}$$

An easy calculation shows that the function $\circ$ defined in this way is associative (for details see e.g. [**?**]). Hence we can substitute in figure 7.6 the $\circ$ gates by the circuit of figure x. One $\circ$ gate now produces a cost of 3 ordinary gates and 2 ordinary gate delays. For the resulting circuit $n - GP$ we conclude

$$d(n - GP) \leq 4 \cdot \log n - 2 \quad \text{and} \quad c(n_G P) \leq 6 \cdot n$$

The point of the construction is, that output i of circuit $n - GP$ computes

$$G_i, P_i) = (g_i, p_i) \circ \ldots \circ (g_0, p_0) = (g_{0,i}, p_{0,i}) = (c_{i+1}, p_{0,i})$$

Thus the circuit from figure 7.8 is an n-adder with cost $O(n)$ and delay $O(logn)$.

TODO figures, copy from Mï¿½ller-Paul but do trivial computation of g_0 as defined in text

7.2 Arithmetic Unit

The symbol of an n-arithmetic unit or short n-AU is shown in Figure 7.9. It is a circuit with the following inputs:

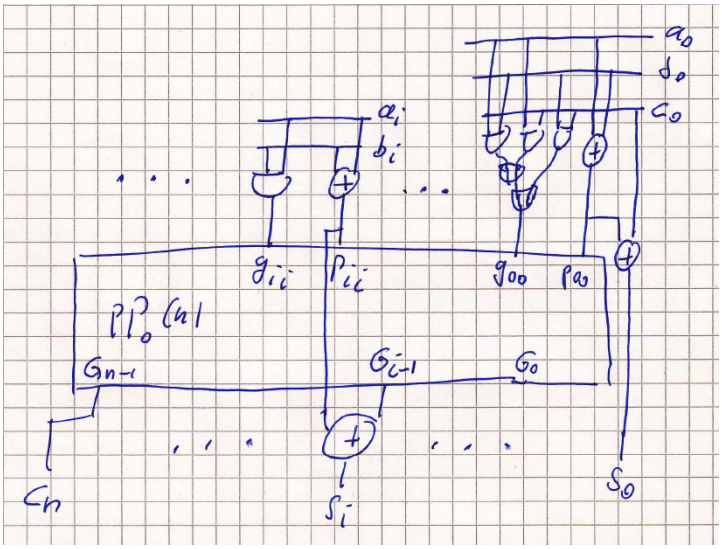


Figure 7.8: Constructing a carry-look-ahead-adder from a parallel prefix circuit for $\circ$

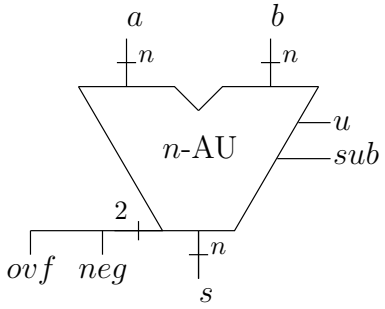


Figure 7.9: Symbol of an n-arithmetic unit.

- operand inputs $a = a[n-1:0], b = b[n-1:0]$ with $a, b \in \mathbb{B}^n$,
- control input $u \in \mathbb{B}$ distinguishing between unsigned (binary) and signed (two's complement) numbers,
- control input $sub \in \mathbb{B}$ indicating whether input b should be subtracted from or added to input a,

and the following outputs:

- result $s[n-1:0] \in \mathbb{B}^n$,
- overflow bit $ovf \in \mathbb{B}$, and
- negative bit $neg \in \mathbb{B}$.

We define the exact result $S \in \mathbb{Z}$ of an arithmetic unit as

$$S = \begin{cases} [a] + [b] & (u, sub) = 00 \\ [a] - [b] & (u, sub) = 01 \\ \langle a \rangle + \langle b \rangle & (u, sub) = 10 \\ \langle a \rangle - \langle b \rangle & (u, sub) = 11. \end{cases}$$

For the result of the ALU, we pick the representative of the exact result in B_n resp. T_n and represent it in the corresponding format

$$s = \begin{cases} twoc_n(S \bmod 2^n) & u = 0 \\ bin_n(S \bmod 2^n) & u = 1, \end{cases}$$

i.e., we have

$$\begin{aligned} [s] &= (S \bmod 2^n) \quad \text{if} \quad u = 0 \\ \langle s \rangle &= (S \bmod 2^n) \quad \text{if} \quad u = 1. \end{aligned}$$

Overflow and negation signals are defined with respect to the exact result. The overflow bit is computed only for the case of two's complement numbers; for binary numbers it is always 0 since the architecture we introduce later does not consider unsigned overflows:

$$\begin{aligned} ovf &\leftrightarrow \begin{cases} S \notin T_n & u = 0 \\ 0 & u = 1 \end{cases} \\ neg &\leftrightarrow S < 0. \end{aligned}$$

Data Paths

The following lemma asserts that, for signed and unsigned numbers, the sum bits s can be computed in exactly the same way:

Lemma 37. *Compute the sum bits as*

$$s = \begin{cases} a +_n b & sub = 0 \\ a -_n b & sub = 1, \end{cases}$$

then

$$\begin{aligned} [s] &= (S \bmod 2^n) \quad \textit{if} \quad u = 0 \\ \langle s \rangle &= (S \bmod 2^n) \quad \textit{if} \quad u = 1. \end{aligned}$$

Proof. For $u = 1$ this follows directly from the definitions. For $u = 0$ we have from Lemma 22 and Lemma 8:

$$\begin{aligned} [s] &\equiv \langle s \rangle \bmod 2^n \\ &\equiv (\begin{cases} \langle a \rangle + \langle b \rangle & sub = 0 \\ \langle a \rangle - \langle b \rangle & sub = 1 \end{cases}) \bmod 2^n \\ &\equiv (\begin{cases} [a] + [b] & sub = 0 \\ [a] - [b] & sub = 1 \end{cases}) \bmod 2^n \\ &\equiv S \bmod 2^n. \end{aligned}$$

From $[s] \in T_n$ and Lemma 11 we conclude

$$[s] = (S \bmod 2^n).$$

□

The main data paths of an n-AU are shown in Figure 7.10. The following lemma asserts that the sum bits are computed correctly.

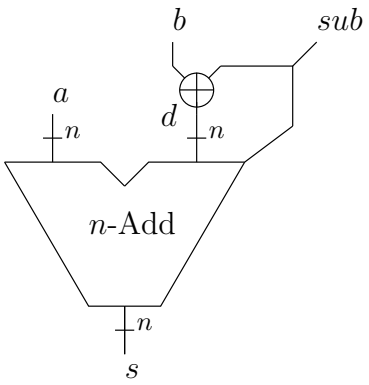


Figure 7.10: Data paths of an n-arithmetic unit.

Lemma 38. *The sum bits $s[n-1:0]$ in Figure 7.10 satisfy*

$$s = \begin{cases} a +_n b & sub = 0 \\ a -_n b & sub = 1. \end{cases}$$

Proof. From the construction of the circuit, we have

$$\begin{aligned} d &= b \oplus sub \\ &= \begin{cases} b & sub = 0 \\ \overline{b} & sub = 1. \end{cases} \end{aligned}$$

From the specification of an n-adder, Lemma 18, and the subtraction algorithm for binary numbers (Lemma 23), we conclude

$$\begin{aligned} \langle s \rangle &= (\left(\begin{cases} \langle a \rangle + \langle b \rangle & sub = 0 \\ \langle a \rangle + \langle \overline{b} \rangle + 1 & sub = 1 \end{cases} \right) \bmod 2^n) \\ &= (\left(\begin{cases} \langle a \rangle + \langle b \rangle & sub = 0 \\ \langle a \rangle - \langle b \rangle & sub = 1 \end{cases} \right) \bmod 2^n). \end{aligned}$$

Application of $bin_n(\cdot)$ to both sides completes the proof of the lemma. □

Negative Bit

We start with the case $u = 0$, i.e. with two's complement numbers. We have

$$\begin{aligned} S &= [a] \pm [b] \\ &= [a] + [d] + sub \\ &\leq 2^{n-1} - 1 + 2^{n-1} - 1 + 1 \\ &= 2^n - 1, \\ S &\geq -2^{n-1} - 2^{n-1} \\ &= -2^n. \end{aligned}$$

Thus,

$$S \in T_{n+1}.$$

According to Lemma 22 we use sign extension to extend operands to $n+1$ bits:

$$\begin{aligned}[a] &= [a_{n-1}a] \\ [d] &= [d_{n-1}d].\end{aligned}$$

We compute an extra sum bit s_n by the basic addition algorithm:

$$s_n = a_{n-1} \oplus d_{n-1} \oplus c_n,$$

and conclude

$$S = [s[n:0]].$$

Again by Lemma 22 this is negative if and only if the sign bit s_n is 1:

$$S < 0 \leftrightarrow s_n = 1.$$

As a result, we have the following lemma.

Lemma 39.

$$u = 0 \rightarrow neg = a_{n-1} \oplus d_{n-1} \oplus c_n$$

For the case $u = 1$, i.e. for binary numbers, a negative result can only occur in the case of subtraction, i.e. if $sub = 1$. In this case we argue along the lines of the correctness proof for the subtraction algorithm:

$$\begin{aligned}S &= \langle a \rangle - \langle b \rangle \\ &= \langle a \rangle - [0b] \\ &= \langle a \rangle + [1\overline{b}] + 1 \\ &= \langle a \rangle + \langle \overline{b} \rangle - 2^n + 1 \\ &= \langle c_n s[n-1:0] \rangle - 2^n \\ &= 2^n(c_n - 1) + \underbrace{\langle s[n-1:0] \rangle}_{\in B_n}.\end{aligned}$$

If $c_n = 1$ we have $S = \langle s \rangle \geq 0$. If $c_n = 0$ we have

$$\begin{aligned}S &= -2^n + \langle s[n-1:0] \rangle \\ &\leq -2^n + 2^n - 1 \\ &= -1.\end{aligned}$$

Thus,

$$u = 1 \rightarrow neg = sub \wedge \overline{c_n},$$

and together with Lemma 39 we get

Lemma 40.

$$\begin{aligned}neg &= \overline{u} \wedge (a_{n-1} \oplus d_{n-1} \oplus c_n) \vee \\ &\quad u \wedge sub \wedge \overline{c_n}\end{aligned}$$

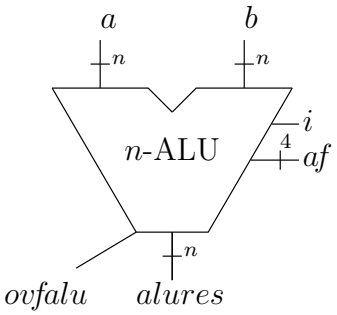


Figure 7.11: Symbol of an n-arithmetic logic unit.

Overflow Bit

We compute the overflow bit only for the case of two's complement numbers, i.e. when $u = 0$. We have

$$\begin{aligned} S &= [a] + [d] + sub \\ &= -2^{n-1}(a_{n-1} + d_{n-1}) + \langle a[n-2:0]\rangle + \langle d[n-2:0]\rangle + sub \\ &= -2^{n-1}(a_{n-1} + d_{n-1}) + \langle c_{n-1}s[n-2:0]\rangle - c_{n-1}2^{n-1} + c_{n-1}2^{n-1} \\ &= -2^{n-1}(a_{n-1} + d_{n-1} + c_{n-1}) + 2^{n-1}(c_{n-1} + c_{n-1}) + \langle s[n-2:0]\rangle \\ &= -2^{n-1}\langle c_n s_{n-1}\rangle + 2^n c_{n-1} + \langle s[n-2:0]\rangle \\ &= -2^n c_n - 2^{n-1}s_{n-1} + 2^n c_{n-1} + \langle s[n-2:0]\rangle \\ &= 2^n(c_{n-1} - c_n) + [s[n-1:0]]. \end{aligned}$$

We claim

$$S \in T_n \leftrightarrow c_{n-1} = c_n.$$

If $c_n = c_{n-1}$ we obviously have $S = [s]$, thus $S \in T_n$. If $c_n = 1$ and $c_{n-1} = 0$ we have

$$-2^n + [s] \leq -2^n + 2^{n-1} - 1 = -2^{n-1} - 1 < -2^{n-1}$$

and if $c_n = 0$ and $c_{n-1} = 1$, we have

$$2^n + [s] \geq 2^n - 2^{n-1} > 2^{n-1} - 1.$$

Thus, in the two latter cases, we have $S \notin T_n$. Because

$$c_n \neq c_{n-1} \leftrightarrow c_n \oplus c_{n-1} = 1,$$

we conclude

Lemma 41.

$$ovf = \overline{u} \wedge c_n \oplus c_{n-1}$$

7.3 Arithmetic Logic Unit (ALU)

Figure 7.11 shows a symbol for the n-ALU constructed here. Width n should be even. The circuit has the following inputs:

- operand inputs $a = a[n-1:0]$, $b = b[n-1:0]$ with $a, b \in \mathbb{B}^n$,
- control inputs $af[3:0] \in \mathbb{B}^4$ and $i \in \mathbb{B}$ specifying the operation that the ALU performs with the operands,

$af[3:0]$	i	$alures[31:0]$	$ovfalu$
0000	*	$a +_n b$	$[a] + [b] \notin T_n$
0001	*	$a +_n b$	0
0010	*	$a -_n b$	$[a] - [b] \notin T_n$
0011	*	$a -_n b$	0
0100	*	$a \wedge b$	0
0101	*	$a \vee b$	0
0110	*	$a \oplus b$	0
0111	0	$\overline{a \vee b}$	0
0111	1	$b[n/2-1:0]0^{n/2}$	0
1010	*	$0^{n-1}([a] < [b]\ ?\ 1 : 0)$	0
1011	*	$0^{n-1}(\langle a \rangle < \langle b \rangle\ ?\ 1 : 0)$	0

Table 7.1: Specification of ALU operations.

and the following outputs:

- result $alures[n-1:0] \in \mathbb{B}^n$,
- overflow bit $ovfalu \in \mathbb{B}$.

The results that must be generated are specified in Table 7.1. There are three groups of operations:

- arithmetic operations,
- logical operations. At first sight, the result $b[n/2:0]0^{n/2}$ might appear odd. This ALU function is later used to compute the upper half of an n bit constant using the immediate fields of an instruction,
- test and set instructions. They compute an n bit result $0^{n-1}z$ where only the last bit is of interest. These instructions can be computed by performing a subtraction in the AU and then testing the negative bit.

Figure 7.12 shows the fairly obvious data paths of an n-ALU. The missing signals are easily constructed. We subtract if $af[1] = 1$. For test and set operations with $af[3] = 1$, the output z is the negative bit neg that we compute for unsigned numbers if $af[0] = 1$ and for signed numbers otherwise. The overflow bit can only differ from zero if we are doing an arithmetic operation. Thus, we have

$$\begin{aligned} sub &= af[1] \\ z &= neg \\ u &= af[0] \\ ovfalu &= ovf \wedge /af[3] \wedge /af[2]. \end{aligned}$$

7.4 Shifter

For $a \in \mathbb{B}^n$ and $i \in [0 : n-1]$ we define the *logical right shift* $srl(n,i)$ of a by n positions by

$$srl(n,i) = 0^i a[n-1:0]$$

Thus operand a is shifted by i position to the right. Unused bits are filled in by zeros.

Let $n = 2^k$ be a power of two. An n-logical right shifter or short $n - SRL$ is a circuit with

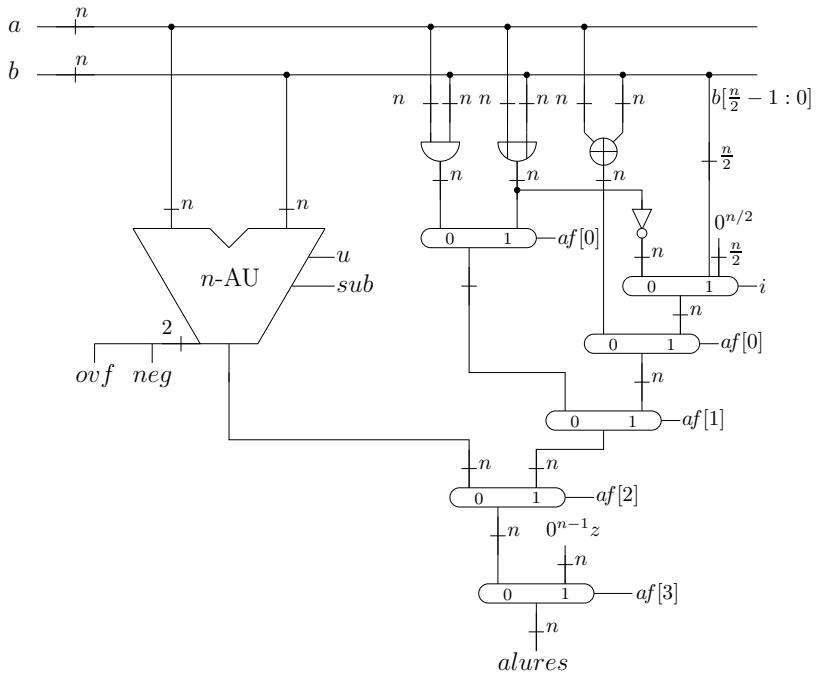


Figure 7.12: Data paths of an n-arithmetic logic unit.

- input $a[n-1:0]$ and $b[k-1:0]$
- an output $sres[n-1:0]$ satisfying

$$shres(a,b) = srl(a, \langle b \rangle)$$

For fixed $i \in [0:n-1]$ the circuit in figure 7.13 satisfies

$$a' = \begin{cases} srl(a,i) & s = 1 \\ a & s = 0 \end{cases}$$

We call it an (n,i)-logical right shifter or short $(n,i) - SRL$. Figure 7.14 shows the construction of n-logical right shifters by a stack of k many $(n,i) - SRL$'s. An obvious induction shows

$$r^{(i)} = sra(n, \langle b[i:0] \rangle)$$

7.5 Branch Condition Evaluation Unit

An n-BCE (see Figure 7.15) has

- inputs $a[n-1:0], b[n-1:0] \in \mathbb{B}^n$,
- inputs $bf[3:0] \in \mathbb{B}^4$ selecting the condition to be tested,
- output $bcres \in \mathbb{B}$ specified by Table 7.2.

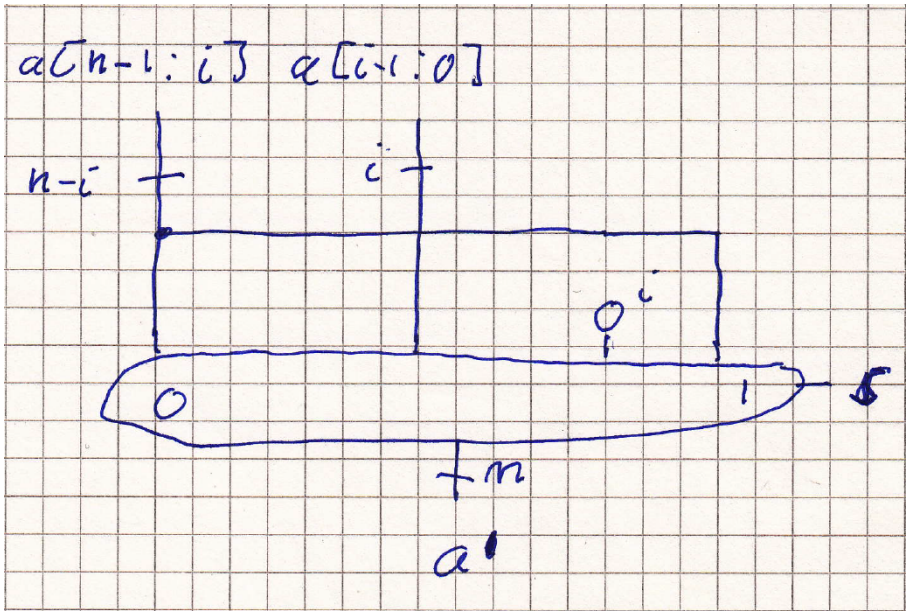


Figure 7.13: Construction of an (n, i)-logical right shifter.

$bf[3:0]$	$bcres$
0010	$[a] < 0$
0011	$[a] \geq 0$
100*	$a = b$
101*	$a \neq b$
110*	$[a] \leq 0$
111*	$[a] > 0$

Table 7.2: Specification of branch condition evaluation.

The auxiliary circuit in Figure 7.16 computes obvious auxiliary signals satisfying

$$\begin{aligned} d &\equiv b \wedge (bf[3] \wedge \overline{bf[2]}) \\ &\equiv \begin{cases} b & bf[3:2] = 01 \\ 0_n & \text{otherwise} \end{cases} \\ eq &\equiv a = d \\ &\equiv \begin{cases} a = b & bf[3:2] = 10 \\ [a] = 0 & \text{otherwise} \end{cases} \\ neq &\equiv \overline{eq} \\ lt &\equiv [a] < 0 \end{aligned}$$

$$le \equiv [a] < 0 \vee \begin{cases} a = b & bf[3:2] = 10 \\ [a] = 0 & \text{otherwise.} \end{cases}$$

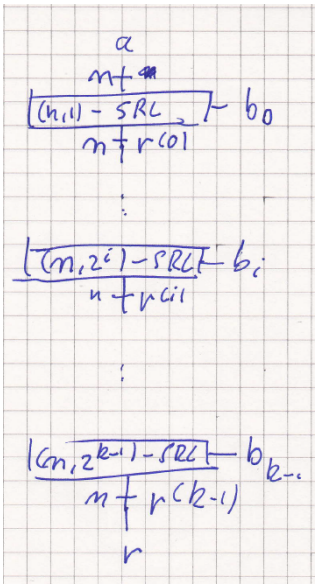


Figure 7.14: Construction of an n-logical right shifter by a stack of $(n, i) - SRL$'s. TODO: label output *shres*

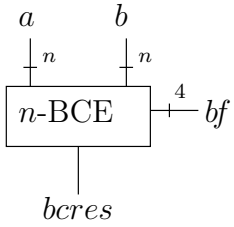


Figure 7.15: Symbol of an n-branch condition evaluation unit.

The result *bcres* can then be computed as

$$\begin{aligned} bcres \quad &\equiv \quad bf[3:1] = 001 \wedge (\overline{bf[0]} \wedge lt \vee bf[0] \wedge \overline{lt}) \\ &\quad\ \vee\ bf[3:2] = 10 \wedge (\overline{bf[1]} \wedge eq \vee bf[1] \wedge \overline{eq}) \\ &\quad\ \vee\ bf[3:2] = 11 \wedge (\overline{bf[1]} \wedge le \vee bf[1] \wedge \overline{le}) \\ &\equiv \quad \overline{bf[3]} \wedge \overline{bf[2]} \wedge bf[1] \wedge (bf[0] \oplus lt) \\ &\quad\ \vee\ bf[3] \wedge \overline{bf[2]} \wedge (bf[1] \oplus eq) \\ &\quad\ \vee\ bf[3] \wedge bf[2] \wedge (bf[1] \oplus le). \end{aligned}$$

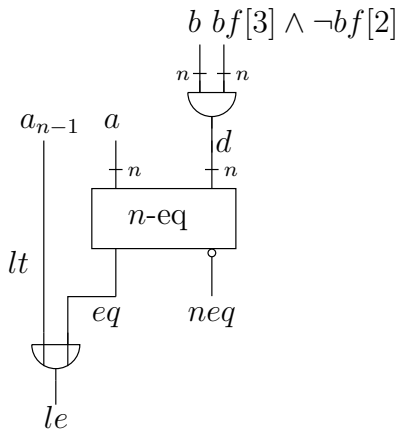


Figure 7.16: Computation of auxiliary signals in an n-branch condition evaluation unit.

Chapter 8

A Basic Sequential MIPS Machine

We define the basic MIPS instruction set architecture (ISA)[1] and present a gate level construction of a processor that implements it. The first Section 8.1 of this chapter is very short. It contains a very compact summary of the instruction set architecture and the assembly language in the form of tables, which define the ISA **if** one knows how to interpret them. In Section 8.2 we provide a succinct and completely precise interpretation of the tables with a small exception: treatment of the three coprocessor instructions and the system call instructions is postponed to the chapter 12 on operating system support. From the mathematical description of the ISA we derive in Section 8.3 the hardware of a sequential - i.e. non pipelined - MIPS processor and argue, that this processor construction is correct.

Because the simple processor construction presented here follows the ISA specification very closely, most of the correctness proof is reduced to very simple simple bookkeeping. Pure bookkeeping ends, where the construction deviates from the ISA or is not implicitly taken from it: i) the implementation of predicates (lemma 44 ii) the use of 30 bit adders in the use of address computations for aligned addresses (lemmas 51 and 52) iii) accesses to memory, where the specification memory is byte addressable and the hardware memory is word addressable (lemmas 42 and 57). In the classroom we treat lemmas concerned in some details and refer for the pure bookkeeping to the lecture notes.

For brevity we treat in this text only one shift operation (srl) and no loads and stores of bytes and half words. A full Implementation of all loads, stores and shifts is not as straight forward as one would expect. The interested reader can find processor constructions for the MIPS instruction set with these instructions in [KMPar].

The final section 8.4 contains some example programs written in MIPS assembly language.

[1] In order to keep the specification simple we make general purpose register 0 an ordinary register, that can be written and read like any other special purpose register. In the original instruction set as documented in [**?**] general purpose register 0 is always 0 and writes to it have no effect.

8.1 Tables

8.1.1 I-Type

In the following table: $m = m_d(ea(c))$ with $ea(c) = rs(c) +_{32} sximm(c)$

opc	Mnemonic	Assembler-Syntax	d	Effect
Data Transfer				
100 011	lw	lw *rt rs imm*	4	rt = m
101 011	sw	sw *rt rs imm*	4	m = rt
Arithmetic, Logical Operation, Test-and-Set				
001 000	addi	addi *rt rs imm*		rt = rs + sxt(imm)
001 001	addiu	addiu *rt rs imm*		rt = rs + sxt(imm)
001 010	slti	slti *rt rs imm*		rt = (rs < sxt(imm) ? 1 : 0)
001 011	sltui	sltui *rt rs imm*		rt = (rs < sxt(imm) ? 1 : 0)
001 100	andi	andi *rt rs imm*		rt = rs ∧ zxt(imm)
001 101	ori	ori *rt rs imm*		rt = rs ∨ zxt(imm)
001 110	xori	xori *rt rs imm*		rt = rs ⊕ zxt(imm)
001 111	lui	lui *rt imm*		rt = imm0^{16}

opc	rt	Mnemonic	Assembler-Syntax	Effect
Branch				
000 001	00000	bltz	bltz *rs imm*	pc = pc + (rs < 0 ? imm00 : 4)
000 001	00001	bgez	bgez *rs imm*	pc = pc + (rs ≥ 0 ? imm00 : 4)
000 100		beq	beq *rs rt imm*	pc = pc + (rs = rt ? imm00 : 4)
000 101		bne	bne *rs rt imm*	pc = pc + (rs ≠ rt ? imm00 : 4)
000 110	00000	blez	blez *rs imm*	pc = pc + (rs ≤ 0 ? imm00 : 4)
000 111	00000	bgtz	bgtz *rs imm*	pc = pc + (rs > 0 ? imm00 : 4)

8.1.2 R-type

opcode	fun	Mnemonic	Assembler-Syntax	Effect
Shift Operation				
000000	000 010	srl	srl *rd rt sa*	rd = srl(rt,sa)
Arithmetic, Logical Operation				
000000	100 000	add	add *rd rs rt*	rd = rs + rt
000000	100 001	addu	addu *rd rs rt*	rd = rs + rt
000000	100 010	sub	sub *rd rs rt*	rd = rs − rt
000000	100 011	subu	subu *rd rs rt*	rd = rs − rt
000000	100 100	and	and *rd rs rt*	rd = rs ∧ rt
000000	100 101	or	or *rd rs rt*	rd = rs ∨ rt
000000	100 110	xor	xor *rd rs rt*	rd = rs ⊕ rt
000000	100 111	nor	nor *rd rs rt*	rd = rs $\overline{\vee}$ rt
Test Set Operation				
000000	101 010	slt	slt *rd rs rt*	rd = (rs < rt ? 1 : 0)
000000	101 011	sltu	sltu *rd rs rt*	rd = (rs < rt ? 1 : 0)
Jumps, System Call				
000000	001 000	jr	jr *rs*	pc = rs
000000	001 001	jalr	jalr *rd rs*	rd = pc + 4 pc = rs
000000	001 100	sysc	sysc	System Call

Coprocessor Instructions					
opcode	fun	rs	Mnemonic	Assembler-Syntax	Effect
010000	011 000	10000	eret	eret	Exception Return
010000		00100	movg2s	movg2s *rd rt*	spr[rd] := gpr[rt]
010000		00000	movs2g	movs2g *rd rt*	gpr[rt] := spr[rd]

8.1.3 J-type

opc	Mnemonic	Assembler-Syntax	Effect
Jumps			
000 010	j	j *iindex*	pc = bin_{32}(pc+4)[31:28]iindex00
000 011	jal	jal *iindex*	R31 = pc + 4, pc = bin_{32}(pc+4)[31:28]iindex00

8.2 MIPS ISA

8.2.1 Configuration and Instruction Fields

A basic *MIPS configuration* c has only three user visible data structures (Figure 8.1):

- $c.pc \in \mathbb{B}^{32}$: the program counter (PC),
- $c.gpr : \mathbb{B}^5 \to \mathbb{B}^{32}$: the general purpose register file consisting of 32 registers, each 32 bits wide. For register addresses $x \in \mathbb{B}^5$ the content of general purpose register x in configuration c is denoted by $c.gpr(x) \in \mathbb{B}^{32}$,
- $c.m : \mathbb{B}^{32} \to \mathbb{B}^8$: the processor memory. It is byte addressable; addresses have 32 bits. Thus, for memory addresses $a \in \mathbb{B}^{32}$ the content of memory location a in configuration c is denoted by $c.m(a) \in \mathbb{B}^8$.

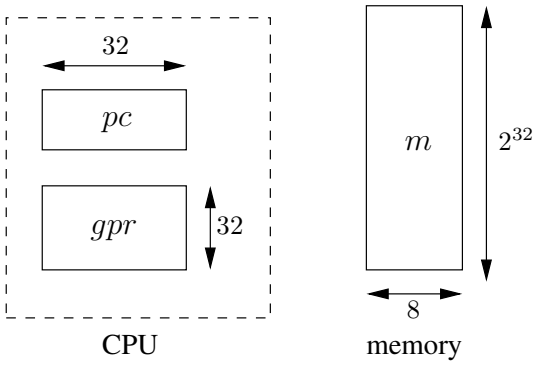


Figure 8.1: Visible data structures of MIPS ISA

Program counter and general purpose registers belong to the central processing unit (CPU).

Let K be the set of all basic MIPS configurations. A mathematical definition of the ISA will be given by a function

$$\delta : K \to K,$$

where

$$c' = \delta(c, reset)$$

is the configuration reached from configuration c, if the next instruction is executed. An ISA computation is a sequence (c^i) of ISA configurations with $i \in \mathbb{N} \setminus \{0\}$ satisfying

$$\begin{aligned} c^1.pc &= 0^{32} \\ c^{i+1} &= \delta(c^i, 0), \end{aligned}$$

i.e. initially the program counter points to address 0^{32} and in each step one instruction is executed. In the remainder of this section we specify the ISA simply by specifying function δ, i.e. by specifying $c' = \delta(c, 0)$ for all configurations c.

Recall, that for numbers $y \in \mathbb{B}^n$ we abbreviate the binary representation of y with n bits as

$$y_n = bin_n(y),$$

e.g. $1_8 = 00000001$ and $3_8 = 00000011$. For memories $m : \mathbb{B}^{32} \to \mathbb{B}^8$, addresses $a \in \mathbb{B}^{32}$ and numbers d of bytes we denote the content of d consecutive memory bytes starting at address a by

$$\begin{aligned} m_1(a) &= m(a) \\ m_{d+1}(a) &= m(a +_{32} d_{32}) \circ m_d(a). \end{aligned}$$

The current instruction $I(c)$ to be executed in configuration c is defined by the 4 bytes in memory addressed by the current program counter:

$$I(c) \equiv c.m_4(c.pc).$$

Because all instructions are 4 bytes long, one requires, that instructions are *aligned* on 4 byte boundaries, or, equivalently that

$$c.pc[1:0] = 00.$$

In case this condition is violated a so called misalignment interrupt is raised.

The six high order bits of the instruction are called the op-code:

$$opc(c) = opc(c)[5:0] = I(c)[31:26].$$

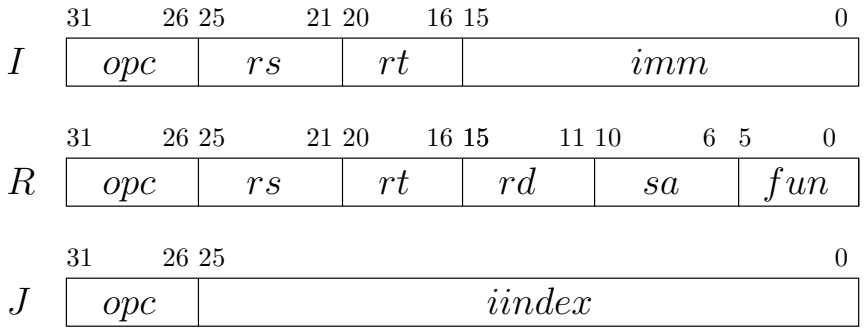


Figure 8.2: Types and fields of MIPS instructions

There are three instruction types: R-, J- and I-type. The instruction type is determined by the following predicates:

$$\begin{aligned} rtype(c) &= opc(c) = 0^*0^4 \\ jtype(c) &= opc(c) = 0^41^* \\ itype(c) &= \overline{rtype(c) \vee jtype(c)}. \end{aligned}$$

Depending on the instruction type, the bits of the instruction are subdivided as shown in Figure 8.2. Addresses of registers in the general purpose register file are specified in the following fields of the instruction:

$$\begin{aligned} rs(c) &= I(c)[25:21] \\ rt(c) &= I(c)[20:16] \\ rd(c) &= I(c)[15:11] \end{aligned}$$

Field

$$sa(I) = I[10:6]$$

specifies the shift distance (shift amount) in the one shift operation (srl) that we implement here.

For R-type instructions, ALU-functions to be applied to the register operands can be specified in the function field:

$$fun(c) = I(c)[5:0].$$

Two kinds of immediate constants can be specified: the immediate constant *imm* in I-type instructions, and an instruction index *iindex* in J-type (like jump) operations:

$$\begin{aligned} imm(c) &\equiv I(c)[15:0] \\ iindex(c) &\equiv I(c)[25:0]. \end{aligned}$$

Immediate constant *imm* has 16 bits. In order to apply ALU functions to it, the constant can be extended with 16 high order bits in two ways: zero extension and sign extension:

$$\begin{aligned} zxtimm(c) &= 0^{16}imm(c) \\ sxtimm(c) &= imm(c)[15]^{16}imm(c) \\ &= I(c)[15]^{16}imm(c). \end{aligned}$$

In case of sign extension, the value of the constant interpreted as a two's complement number does not change:

$$[sxtimm(c)] = [imm(c)].$$

8.2.2 Instruction Decoding

For every mnemonic mn of a MIPS instruction from the tables above we define a predicate $mn(c)$ which is true, if $I(c)$ is an mn instruction. For instance

$$\begin{aligned} l(c) &\equiv opc(c) = 100011 \\ bltz(c) &\equiv opc(c) = 0^5 1 \wedge rt(c) = 0^5 \\ add(c) &\equiv rtype(c) \wedge fun(c) = 10^5. \end{aligned}$$

The remaining predicates directly associated to the mnemonics of the assembly language are derived in the same way from the tables. We group the basic instruction set into 5 groups and define for each group a predicate that holds, if an instruction from that group is to be executed:

- ALU-operations of I-type are recognized by the leading three bits of the opcode, resp. $I(c)[31:29]$; ALU-operations of R-type - by the two leading bits of the function code, resp. $I(c)[5:4]$:

$$\begin{aligned} alur(c) &\equiv rtype(c) \wedge fun(c)[5:4] = 10 \\ &\equiv rtype(c) \wedge I(c)[5:4] = 10 \\ alui(c) &\equiv itype(c) \wedge opc(c)[5:3] = 001 \\ &\equiv itype(c) \wedge I(c)[31:29] = 001 \\ alu(c) &\equiv alur(c) \vee alui(c) \end{aligned}$$

- loads and stores are of I-type and are recognized by the three leading bits of the opcode:

$$\begin{aligned} l(c) &\equiv opc(c)[5:3] = 100 \\ &\equiv I(c)[31:29] = 100 \\ s(c) &\equiv opc(c)[5:3] = 101 \\ &\equiv I[31:29] = 101 \\ ls(c) &\equiv l(c) \vee s(c) \\ &\equiv opc(I)[5:4] = 10 \\ &\equiv I(c)[31:30] = 10 \end{aligned}$$

- branches are of I-Type and are recognized by the three leading bits of the opcode:

$$\begin{aligned} b(c) &\equiv itype(c) \wedge opc(c)[5:3] = 000 \\ &\equiv itype(c) \wedge I(c)[31:29] = 000 \end{aligned}$$

We define jumps in a brute force way:

$$\begin{aligned} jump(c) &\equiv jr(c) \vee jalr(c) \vee j(c) \vee jal(c) \\ jb(c) &\equiv jump(c) \vee b(c). \end{aligned}$$

8.2.3 ALU-Operations

We can now go through the ALU-operations in the tables one by one and give them precise interpretations. We do this for two examples:

add(c): The table specifies the effect as $rd = rs + rt$. This is to be interpreted as the corresponding register contents: on the right hand side of the equation for c, i.e. before execution of the instruction; on the left hand side for c':

$$c'.gpr(rd(c)) = c.gpr(rs(c)) +_{32} c.gpr(rt(c)).$$

Other register contents and the memory content do not change:

$$\begin{aligned} c'.gpr(x) &= c.gpr(x) \quad \text{for} \quad x \neq rd(c) \\ c'.m &= c.m. \end{aligned}$$

The program counter is advanced by four bytes to the next instruction:

$$c'.pc = c.pc +_{32} 4_{32}.$$

addi(c): The second operand is now the sign extended immediate constant:

$$\begin{aligned} c'.gpr(x) &= \begin{cases} c.gpr(rs(c)) +_{32} sxtimm(c) & x = rt(c) \\ c.gpr(x) & \text{otherwise} \end{cases} \\ c'.m &= c.m \\ c'.pc &= c.pc +_{32} 4_{32}. \end{aligned}$$

It is clear how to derive precise specifications for the remaining ALU-operations, but we take a shortcut exploiting the fact that we have already constructed an ALU that was specified in Table 7.1.

This table defines functions $alures(a, b, af, i)$ and $ovf(a, b, af, i)$. As we do not treat interrupts (yet) we use only the first of these functions here. We observe that in all ALU operations a function of the ALU is performed. The left operand is always

$$lop(c) \equiv c.gpr(rs(c)).$$

For R-type operations, the right operand is the register specified by the rt field of R-type instructions. For I-type instructions it is the sign extended immediate operand if $opc(c)[2] = I(c)[28] = 0$ or zero extended immediate operand if $opc(c)[2] = 1$. [2] Thus, we define immediate fill bit $ifill(c)$, extended immediate constant $xtimm(c)$, and right operand $rop(c)$ in the following way:

$$\begin{aligned} ifill(c) &\equiv \begin{cases} 0 & opc(c)[3:2] = 11 \\ imm(c)[15] & otherwise \end{cases} \\ &= I(c)[15] \wedge \overline{(I(c)[29] \wedge I(c)[28])} \\ xtimm(c) &\equiv \begin{cases} sxtimm(c) & opc(c)[2] = 0 \\ zxtimm(c) & opc(c)[2] = 1 \end{cases} \\ &= ifill(c)^{16} imm(c) \\ rop(c) &\equiv \begin{cases} c.gpr(rt(c)) & rtype(c) \\ xtimm(c) & \text{otherwise.} \end{cases} \end{aligned}$$

[2] Note that for instructions *addiu* and *sltiu* this is counterintuitive. The letter u in the instruction name suggests unsigned arithmetic, and arithmetic with binary numbers is indeed performed by these operations. But the immediate operand for these instructions is sign extended and not zero extended, thus we have

$$\langle sextimm(c) \rangle \neq \langle imm(c) \rangle \quad \text{if} \quad imm(c)[15] = 1$$

The instruction set manual [**?**] acknowledges this in the documentation of the addiu instruction and calls the term 'unsigned' in the instruction name a misnomer.

Comparing Table 7.1 with the tables for I-type and R-type instructions we see that bits $af[2:0]$ of the ALU control can be taken from the low order fields of the opcode for I-type instructions and from the low order bits of the function field for R-type instructions:

$$\begin{aligned} af(c)[2:0] &\equiv \begin{cases} fun(c)[2:0] & rtype(c) \\ opc(c)[2:0] & \text{otherwise} \end{cases} \\ &= \begin{cases} I(c)[2:0] & rtype(c) \\ I(c)[28:26] & \text{otherwise.} \end{cases} \end{aligned}$$

For bit $af[3]$ things are more complicated. For R-type instructions it can be taken from the function code. For I-type instructions it must only be forced to 1 for the two test and set operations, which can be recognized by $opc(c)[2:1] = 01$:

$$\begin{aligned} af(c)[3] &\equiv \begin{cases} func(c)[3] & rtype(c) \\ \overline{opc(c)[2]} \wedge opc(c)[1] & \text{otherwise} \end{cases} \\ &\equiv \begin{cases} I(c)[3] & rtype(c) \\ \overline{I(c)[28]} \wedge I(c)[27] & \text{otherwise.} \end{cases} \end{aligned}$$

The i-input of the ALU distinguishes for $af[3:0] = 1111$ between the *lui*-instruction of I-type for $i = 0$ and the *nor*-instruction of R-type for $i = 1$. Thus we set it to $itype(c)$. The result of the ALU computed with these inputs is denoted by

$$ares(c) \equiv alures(lop(c), rop(c), af(c), itype(c)).$$

Depending on instruction type the destination register $rdes$ is specified by the rd field or the rt field:

$$rdes(c) \equiv \begin{cases} rd(c) & rtype(c) \\ rt(c) & \text{otherwise.} \end{cases}$$

A summary of *all* ALU operations is then

$$\begin{aligned} alu(c) &\rightarrow \\ c'.gpr(x) &= \begin{cases} ares(c) & x = rdes(c) \\ c.gpr(x) & \text{otherwise} \end{cases} \\ c'.m &= c.m \\ c'.pc &= c.pc +_{32} 4_{32}. \end{aligned}$$

8.2.4 Shift

We implement only a single shift operation. The shift distance is taken from the shift amount field $sa(c)$. The left operand $slop(c)$ to be shifted is always the register specified by the rt field

$$slop(c) = c.gpr(rt(c))$$

The result of the shifter from section **??** with these operands is

$$sres(c) = shres(slop(c), sa(c))$$

The destination register is specified by the rd field. We summarize

$$\begin{aligned} srl(c) &\rightarrow \\ c'.gpr(x) &= \begin{cases} sres(c) & x = rd(c) \\ c.gpr(x) & \text{otherwise} \end{cases} \\ c'.m &= c.m \\ c'.pc &= c.pc +_{32} 4_{32} \end{aligned}$$

8.2.5 Branch and Jump

A branch condition evaluation unit was specified in Table 7.2. It computes a function $bcres(a, b, bf)$. We use this function with the following parameters:

$$\begin{aligned} blop(c) &\equiv c.gpr(rs(c)) \\ brop(c) &\equiv c.gpr(rt(c)) \\ bf(c) &\equiv opc(c)[2:0] \circ rt(c)[0] \\ &= I(c)[28:26]I[16]. \end{aligned}$$

and define the result of a branch condition evaluation as

$$bres(c) \equiv bcres(blop(c), brop(c), bf(c)).$$

The next program counter $c'.pc$ is usually computed as $c.pc +_{32} 4_{32}$. This order is only changed in jump instructions or in branch instructions, where the branch is taken, i.e. the branch condition evaluates to 1. We define

$$jbtaken(c) \equiv jump(c) \vee b(c) \wedge bres(c).$$

In case of a jump or a branch taken, there are three possible jump targets

Branch instructions involve a *relative* branch. The PC is incremented by a branch distance:

$$\begin{aligned} b(c) \wedge bcres(c) &\rightarrow \\ bdist(c) &= imm(c)[15]^{14} imm(c)00 \\ btarget(c) &= c.pc +_{32} bdist(c). \end{aligned}$$

Note, that the branch distance is a kind of a sign extended immediate constant, but due to the alignment requirement the low order bits of the jump distance must be 00. Thus, one uses the 16 bits of the immediate constant for bits $[17:2]$ of the jump distance. Sign extension is used for the remaining bits. Note also that address arithmetic is modulo 2^n. We have

$$\begin{aligned} \langle c.pc \rangle + \langle bdist(c) \rangle &\equiv [c.pc] + [bdist(c)] \bmod 2^n \\ &= [c.pc] + [imm(c)00]. \end{aligned}$$

Thus, backward jumps are realized with negative $[imm(c)]$.

R-type jumps for instructions jr and $jalr$. The branch target is specified by the rs field of the instruction:

$$\begin{aligned} jr(c) \vee jalr(c) &\rightarrow \\ btarget(c) &= c.gpr(rs(c)). \end{aligned}$$

J-Type jumps for instructions j and jal. The branch target is computed in a rather peculiar way: i) the PC is incremented by 4. *Then* bits $[27:0]$ are replaced by the $iindex$ field of the instruction:

$$\begin{array}{rcl} j(c) \vee jal(c) & \rightarrow & \\ btarget(c) & = & (c.pc +_{32} 4_{32})[31:28]iindex(c)00 \end{array}$$

Now we can define the next PC computation for *all* instructions as

$$\begin{array}{rcl} btarget(c) & \equiv & \begin{cases} c.pc +_{32} imm(c)[15]^{14} imm(c)00 & b(c) \\ c.gpr(rs(c)) & jr(c) \vee jalr(c) \\ (c.pc +_{32} 4_{32})[31:28]iindex(c)00 & \text{otherwise.} \end{cases} \\ c'.pc & = & \begin{cases} btarget(c) & jbtaken(c) \\ c.pc +_{32} 4_{32} & \text{otherwise.} \end{cases} \end{array}$$

Jump and Link. The two jump instructions jal and $jalr$ are used to implement calls of procedures. Besides setting the PC to the branch target they prepare the so called *link address* by saving the incremented PC

$$linkad(c) = c.pc +_{32} 4_{32}$$

in a register. For the R-type instruction $jalr$ this register is specified by the rd field. J-type instruction jal does not have an rs field, and the incremented PC is stored in register 31 ($= \langle 1^5 \rangle$). Branch and jump instructions do not change the memory.

For the update of registers in branch and jump instructions we therefore have

$$\begin{array}{rcl} bj(c) & \rightarrow & \\ c'.gpr(x) & = & \begin{cases} linkad(c) & jalr(c) \wedge x = rd(c) \vee jal(c) \wedge x = 1^5 \\ c.gpr(x) & \text{otherwise} \end{cases} \\ c'.m & = & c.m. \end{array}$$

8.2.6 Loads and Stores

A *byte* is a string $x \in \mathbb{B}^8$. Let $n = 8 \cdot k$ be a multiple of 8, let $a \in \mathbb{B}^n$ be a string consisting of k bytes. For $i \in [k-1:0]$ we define byte i of string a as

$$byte(i, a) = a[8 \cdot (i+1) - 1 : 8 \cdot i].$$

The load and store operations of the basic MIPS instruction set each access four of bytes of memory starting at a so called *effective address* $ea(c)$. Addressing is always relative to a register specified by the rs field. The offset is specified by the immediate field:

$$ea(c) = c.gpr(rs(c)) +_{32} sximm(c).$$

Note, that the immediate constant is sign extended, thus negative offsets can be realized in the same way as negative branch distances. Effective addresses are required to be *aligned*. If we interpret them as binary numbers they have to be divisible by 4:

$$4 \mid \langle ea(c) \rangle$$

or, equivalently,

$$ea(c)[1:0] = 00.$$

If this condition is violated a misalignment interrupt mal is raised. The treatment of interrupts are postponed to chapter **??**.

Stores. A store instruction takes register specified by the rt field and stores it in the 4 consecutive bytes of memory starting at address $ea(c)$ $m_{d(c)}(ea(c))$. Other memory bytes and register values are not changed. The PC is incremented by 4 (but we have already defined that).

$$\begin{aligned} s(c) &\rightarrow \\ c'.m(x) &= \begin{cases} byte(i, c.gpr(rt(c))) & x = ea(c) +_{32} i_{32} \wedge i < 4 \\ c.m(x) & \text{otherwise} \end{cases} \\ c'.gpr &= c.gpr \end{aligned}$$

[3]

Loads. Loads like stores access 4 bytes of memory starting at address $ea(c)$. The result is stored in the destination register, which is specified by the rt field of the instruction. The load result $lres(c) \in \mathbb{B}^{32}$ is computed as

$$lres(c) = c.m_4(ea(c))$$

The general purpose register specified by the rt field is updated. Other registers and the memory are left unchanged:

$$\begin{aligned} l(c) &\rightarrow \\ c'.gpr(x) &= \begin{cases} lres(c) & x = rt(c) \\ c.gpr(x) & \text{otherwise} \end{cases} \\ c'.m &= c.m. \end{aligned}$$

[3] A word of caution in case you plan to enter this into a CAV system: the first case of the "definition" of $c'.m(x)$ is very well understandable for humans, but actually it is a shorthand for the following: if

$$\exists i : x = ea(c) +_{32} i_{32}$$

then update $c.m(x)$ with the hopefully unique i satisfying this condition. In this case we can compute this i by solving the equation

$$x = ea(c) +_{32} i_{32}$$

resp.

$$\langle x \rangle = (\langle ea(c) \rangle + i \bmod 2^{32}).$$

From alignment we conclude

$$\langle ea(c) \rangle + i \leq 2^{32} - 1.$$

Hence

$$(\langle ea(c) \rangle + i \bmod 2^{32}) = \langle ea(c) \rangle + i.$$

And we have to solve

$$\langle x \rangle = \langle ea(c) \rangle + i$$

as

$$i = \langle x \rangle - \langle ea(c) \rangle.$$

This turns the above definition into

$$c'.m(x) = \begin{cases} byte(\langle x \rangle - \langle ea(c) \rangle, c.gpr(rt(c))) & \langle x \rangle - \langle ea(c) \rangle \in [0 : d(c) - 1] \\ c.m(x) & \text{otherwise,} \end{cases}$$

which is not so readable for humans.

8.2.7 ISA Summary

We collect all previous definitions of destination registers for the general purpose register file into

$$cad(c) = \begin{cases} 1^5 & jal(c) \\ rd(c) & rtype(c) \\ rt(c) & \text{otherwise.} \end{cases}$$

Also we collect the data *gprin* to be written into the general purpose register file. For technical reasons we define on the way an intermediate result C:

$$\begin{aligned} C(c) &= \begin{cases} linkad(c) & jal(c) \vee jalr(c) \\ sres(c) & srl(c) \\ ares(c) & \text{otherwise} \end{cases} \\ gprin(c) &= \begin{cases} lres(c) & l(c) \\ C(c) & \text{otherwise.} \end{cases} \end{aligned}$$

Finally we collect in a general purpose register write signal all situations, when some general purpose register is updated:

$$gprw(c) \equiv alu(c) \vee srl(c) \vee l(c) \vee jal(c) \vee jalr(c).$$

Now we can summarize the MIPS ISA in three rules concerning the updates of PC, general purpose registers and memory:

$$\begin{aligned} c'.pc &= \begin{cases} btarget(c) & jbtaken(c) \\ c.pc +_{32} 4_{32} & \text{otherwise} \end{cases} \\ c'.gpr(x) &= \begin{cases} gprin(c) & x = cad(c) \wedge gprw(c) \\ c.gpr(x) & \text{otherwise} \end{cases} \\ c'.m(x) &= \begin{cases} byte(i, c.gpr(rs(c))) & x = ea(c) +_{32} i_{32} \wedge i < 4 \wedge s(c) \\ c.m(x) & \text{otherwise.} \end{cases} \end{aligned}$$

8.3 A Sequential Processor Design

From the ISA spec we derive a hardware implementation of the basic MIPS processor. It will execute every MIPS instruction in two hardware cycles: in a *fetch* cycle, the hardware machine fetches an instruction from memory, in an *execute* cycle, the machine executes the current instruction.

In order to prove correctness of the hardware implementation with respect to the MIPS ISA specification, we prove simulation as follows: Given a hardware computation

$$h^0, h^1, \ldots, h^{2t}, h^{2t+1}, h^{2t+2}, \ldots$$

we show that there is a corresponding ISA computation

$$c^0, c^1, \ldots, c^t, c^{t+1}, \ldots$$

in such a way that a simulation relation

$$c^i \sim h^{2i}$$

holds for all $i \in \mathbb{N}$.

We proceed by giving the hardware construction and simulation relation, arguing that the given construction is indeed correct at the time we introduce individual hardware components.

8.3.1 Hardware Configuration

A hardware configuration h of the MIPS implementation contains among others the following components:

- program counter register $h.pc \in \mathbb{B}^{32}$,
- general purpose register RAM $h.gpr : \mathbb{B}^5 \to \mathbb{B}^{32}$,
- *word addressable* $(32, r, 30)$-RAM-ROM $h.m : \mathbb{B}^{30} \to \mathbb{B}^{32}$,

More components will be specified shortly.

Recall that the ISA specification states that the hardware implementation only needs to work if all memory accesses of the ISA computation (c^i) are *aligned*, i.e.

$$\forall i > 0 : c^i.pc[1:0] = 00 \wedge \\ ls(c^i) \to ea(c^i)[1:0] = 00).$$

Now consider that the RAM-ROM we use as memory of our hardware machine is word-addressable, in contrast to the byte-addressable memory of our ISA specification. Having a word-addressable hardware memory has the advantage that we can serve any properly aligned memory access up to word-size in a single cycle. The reason why the hardware memory contains a ROM portion was already explained in section 6.3: we need after power up/reset some known portion of memory, where the boot loader resides.

We define the simulation relation $c \sim h$ which states that hardware configuration h encodes ISA configuration c by

$$c \sim h \equiv h.pc = c.pc \wedge h.gpr = c.gpr \wedge c.m \sim_M h.m$$

where the simulation relation $\sim_M$ for the memory is given as the following simple embedding:

$$c.m \sim_M d.m \equiv \forall a \in \mathbb{B}^{32} : c.m_4(a00) = h.m(a)$$

i.e. for word addresses a, $h.m(a)$ contains the four bytes of ISA memory starting at address $a00$.

In order to prove hardware construction correctness, we perform induction on the number of steps of the ISA specification machine: We need to show for initial configurations c^0 and h^0 that $c^0 \sim h^0$ holds (we assume that in cycle -1 the reset signal is active). In the induction step, we show that, given c^i and h^{2i} with $c^i \sim h^{2i}$ the simulation relation is maintained when we perform two steps of the hardware machine and a single step of the ISA specification machine, resulting in $c^{i+1} \sim h^{2(i+1)}$.

Since we use a RAM-ROM to implement the memory of the machine, there is another software condition that we need to obey in order to be able to maintain the memory simulation relation:

$$\forall i : s(c^i) \to ea(c^i)[31:(r+2)] \neq 0^{32-(r+2)}$$

That is, every write access to the memory must go to an address that does not belong to the ROM, since writes to the ROM do not have any effect (recall that ROM stands for read-only-memory).

We only prove hardware construction correctness for executions of the ISA that obey our software conditions.

Definition 20 (Hardware Correctness Software Conditions)**.** *The software conditions on ISA computation $(c)_i$ are:*

1. *all memory accesses are aligned properly:*

$$\forall i > 0 : c^i.pc[1:0] = 00 \wedge \\ ls(c^i) \to ea(c^i)[1:0] = 00).$$

2. *there are no writes to the ROM portion of memory:*

$$\forall i : s(c^i) \to ea(c^i)[31:(r+2)] \neq 0^{32-(r+2)}$$

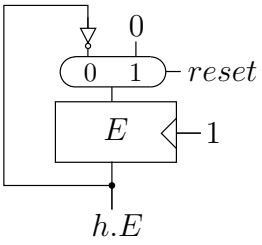


Figure 8.3: Computation of the execute signal E.

8.3.2 Fetch and Execute Cycles

Since a single MIPS instruction may perform up to two memory accesses and our main memory can only serve a single request per cycle, we construct the hardware machine in such a way that it performs *fetch* cycles and *execute* cycles in an alternating fashion. This is done by introducing register E ($E = 1$ stands for *execute*) and the surrounding the circuitry given in figure 8.3. Recall that this was the introductory example when we introduced the model of clocked circuits. Formally, we have to include

- $$h.E \in \mathbb{B}$$

  as component of the hardware configuration. In section 5.3 we already showed

$$h^t.E = t \bmod 2$$

Even cycles t (with $h^t.E = 0$) will be called *fetch cycles* and odd cycles (with $h^t.E) = 1$) will be called *execute cycles*.

The portion of the hardware relevant for reset and instruction fetch is shown in figure 8.4. Portions with the three dots will be filled in later. The *nextpc* circuit computing the next program counter will be discussed in Subsection 8.3.8.

8.3.3 Reset

Recall from the clocked circuit model that the reset signal is active in cycle $t = -1$ and zero afterwards. The pc is clocked during cycle -1 and thus initially, i.e. after reset, we have

$$h^0.pc = 0^{32} = c^0.pc.$$

Hence the first part of the simulation relation holds for $i = 0$. We take no precautions to prevent writes to $h.gpr$ or $h.dm$ during cycle -1 and define

$$\begin{aligned} c^0.gpr &= h^0.gpr \\ c^0.m_4(a00) &= h^0.m(a) \end{aligned}$$

We can conclude

$$c^0 \sim h^0$$

From now on, let $i > 0$ and assume $c^i \sim h^{2i}$. We construct the hardware in such a way that we can conclude $c^{i+1} \sim h^{2i}$.

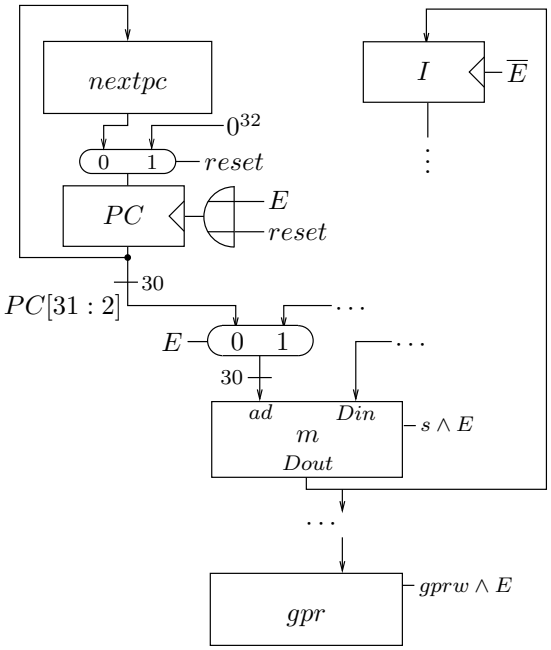


Figure 8.4: Portion of the MIPS hardware relevant for reset and instruction fetch. TODO: rename Dout to mout

8.3.4 Instruction Fetch

At the end of fetch cycles, instructions fetched from memory are clocked into a register I of the hardware, which is called the instruction register. Formally it has to be included as a component of the hardware configuration

- $$c.I \in \mathbb{B}^{32}$$

For the hardware in figure 8.4 we will show: if in configuration h reset if off ($reset(h) = 0$, a fetch is performed ($h.E = 0$) and hardware configuration h codes c i.e. $h \sim c$, then in the next hardwrre configuration h' the simulation still holds and the ISA instruction $I(c)$ has been clocked into the instruction register

Lemma 42 (Correctness of fetch cycles)**.**

$$reset(h) = 0 \wedge h.E = 0 \wedge c \sim h \rightarrow c \sim h' \wedge h'.I == (c)$$

This of course implies for the induction step of the processor correctness proof

$$c^i \sim c^i \wedge h^{2i+1}.I$$

Inputs for the instruction register comes from the data output $mout$ of memory m

$$Iin(h) = mout(h)$$

Because $h.E = 0$ the hardware memory is addressed with bits

$$h.pc[31 : 2]$$

From $c.m \sim_M h.m$, we have

$$h.m(h.pc[31 : 2]) = c.m_4(h.pc[31 : 2]00)$$

and from $c \sim h$, we obtain

$$h.pc[31:2] \;=\; c.pc[31:2].$$

We conclude that the hardware instruction $Iin(h)$ fetched by the circuitry in Figure 8.4 is

$$\begin{aligned} Iin(h) &= mout(h.m) \\ &= h.m(h.pc[31:2]) \\ &= c.m_4(c.pc[31:2]00) \\ &= c.m_4(c.pc) \quad \text{(alignment)} \\ &= I(c). \end{aligned}$$

Note that, by construction, the instruction fetched from memory is stored in the instruction register $h.I$ at the end of the cycle, since for the clock enable signal Ice of the instruction register we have

$$Ice(h) = \sim h.E = 1$$

. Thus, we have

$$h'.I = I(c).$$

The main memory and the general purpose register file are written under control of hardware signals $E \wedge s$ resp. $E \wedge gprw$. Hardware signals s and w will be defined later as the obvious counterparts of the ISA predicates with the same name. Without knowing their exact definition we can already conclude that hardware memory and hardware gpr can only be written in execute cycles. Also, the program counter of the hardware is only clocked in execute cycles. Thus we have

$$\begin{aligned} h'.gpr &= h.gpr \\ &= c.gpr \quad \text{(induction hypothesis)} \end{aligned}$$

as well as

$$\begin{aligned} h'.m &= h.m \\ &\sim_M c.m \quad \text{(induction hypothesis)} \end{aligned}$$

Finally, in fetch cycles the pc is not updated and we get

$$\begin{aligned} h'.pc &= h.pc \\ &= c.pc \quad \text{(induction hypothesis)} \end{aligned}$$

This completes the proof of lemma 42

8.3.5 Proof Goals for the Execute Stage

In the analysis of the subsequent execute cycle $2i+1$ we assume Lemma **??** and need to establish $c^{2i+2} \sim h^{2i+2}$. For numerous functions and predicates $f(c)$ which are already defined as functions of MIPS ISA configurations c we have to consider their obvious counter parts $f_h(d)$, which are functions of hardware configurations d. In order to avoid cluttered notation caused by the subscript h we often overload the notation and use for both functions the same name f. Confusion will not arise, because we will use such functions f only with arguments c or h, where MIPS configurations are denoted by c and hardware configurations by d.

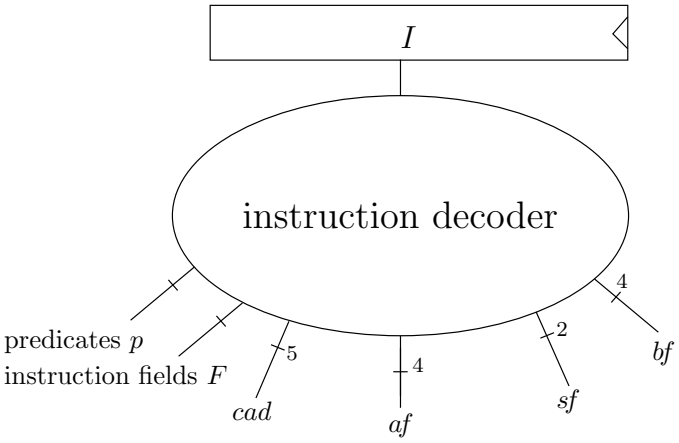


Figure 8.5: Instruction decoder

For the remainder of this chapter we will consider hardware configurations h and ISA configurations c where reset is off, an execute cycle is performed ($h.E = 1$), hardware configuration h codes c i.e. $h \sim c$, and the instruction register $h.I$ contains the ISA instruction $I(c)$

$$reset(h) = 0 \wedge h.E = 1 \wedge c \sim h \wedge h.I = I(c) \tag{8.1}$$

Our general goal will be to establish for ISA functions $f(c)$ and their counter parts $f(h)$ in the hardware

$$f(h) = f(c)$$

Hardware will be constructed such that we can show: if condition **??** holds, then after clocking the next hardware configuration h' codes the the next ISA configuration c'

Lemma 43 (Correctness of execute cycles)**.**

$$reset(h) = 0 \wedge h.E = 1 \wedge c \sim h \wedge H.I = I(c) \rightarrow c' \sim h'$$

For the induction step of the processor correctness proof, we know that the hypothesis of this lemma is fulfilled for $h = h^{2i+1}$ and $c = c^i$ and we can conclude

$$c^{i+1} \sim h^{2i+2}$$

8.3.6 Instruction Decoder

The instruction decoder shown in Figure 8.5 computes the hardware version of functions $f(c)$ that only depend on the current instruction $I(c)$, i.e. which can be written as

$$f(c) = f'(I(c)).$$

For example

$$\begin{aligned} rtype(c) &\equiv opc(c) = 0^*0^4 \\ &\equiv I(c)[31:26] = 0^*0^4 \\ rtype'(I[31:0]) &\equiv I[31:26] = 0^*0^4 \end{aligned}$$

or

$$\begin{aligned} rd(c) &= I(c)[15:11] \\ rd'(I[31:7]) &= I[15:11] \end{aligned}$$

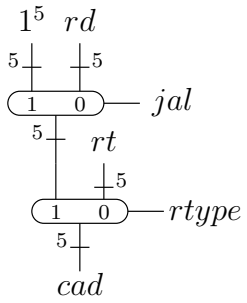


Figure 8.6: C address computation

Predicates. Let p be an ISA predicate. By lemma 31 there is a circuit C with inputs $I[31:1]$ of the hardware instruction register and a gate g in the circuit, such that

$$g(I) = p'(I)$$

Connecting inputs I to the instruction register $h.I$ of the hardware gives

$$g(h) = g(h.I) = p'(h.I)$$

Condition 8.1 gives

$$\begin{aligned} g(h) &= p'(h.I) \\ &= p'(I(c)) \\ &= p(c) \end{aligned}$$

Renaming hardware signal g to p gives

Lemma 44. *For all predicates p:*

$$p(h) = p(c)$$

Instruction Fields. All instruction fields F have the form

$$F(c) = I(c)[m:n].$$

Compute the hardware version as

$$F(h) = h.I[m:n]$$

Condition 8.1 gives

$$\begin{aligned} F(h) &= h.I[m:n] \\ &= I(c)[m:n] \\ &= F(c) \end{aligned}$$

Thus we have

Lemma 45. *For all function fields F:*

$$F(h) = F(c)$$

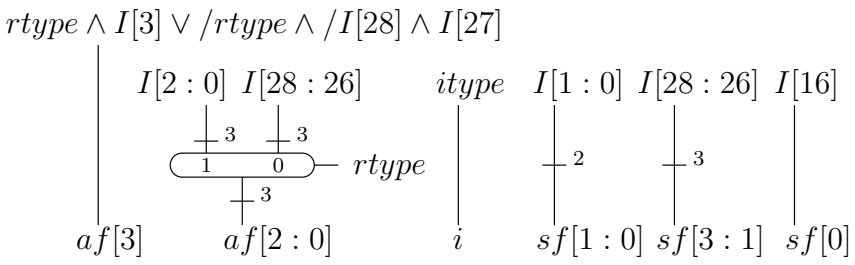


Figure 8.7: Computation of function fields for ALU, and BCE TODO: remove sf

C Address. The output $cad(h)$ in Figure 8.6 computes the C address for the general purpose register file. Using lemmas 44 and 45 we have

$$\begin{aligned} cad(h) &= \begin{cases} 1^5 & jal(h) \\ rd(h) & rtype(h) \\ rt(h) & \text{otherwise} \end{cases} \\ &= \begin{cases} 1^5 & jal(c) \\ rd(c) & rtype(c) \\ rt(c) & \text{otherwise} \end{cases} \\ &= cad(h) \end{aligned}$$

Thus we have

Lemma 46.

$$cad(h) = cad(c)$$

Extended immediate constant. The fill bit $ifill(c)$ is a predicate and $imm(c)$ is a field of the instruction. Thus, we can compute the extended immediate constant in hardware as

$$\begin{aligned} xtimm(h) &= ifill(h)^{16} imm(h) \\ &= ifill(c)^{16} imm(c) \quad \text{(Lemmas 44 and 45)} \\ &= xtimm(c). \end{aligned}$$

Hence we have

Lemma 47.

$$xtimm(h) = xtimm(c)$$

Function fields for ALU, SU, and BCE. Figure 8.7 shows the computation of the function fields *af*,*i*, *sf*, and *bf* for the ALU, the shift unit, and the branch condition evaluation unit.

Outputs $af(h)[2:0]$ satisfy by Lemmas 44 and 45

$$\begin{aligned} af(h)[2:0] &= \begin{cases} I(h)[2:0] & rtype(h) \\ I(h)[28:26] & \text{otherwise} \end{cases} \\ &= \begin{cases} I(c)[2:0] & rtype(c) \\ I(c)[28:26] & \text{otherwise} \end{cases} \\ &= af(c). \end{aligned}$$

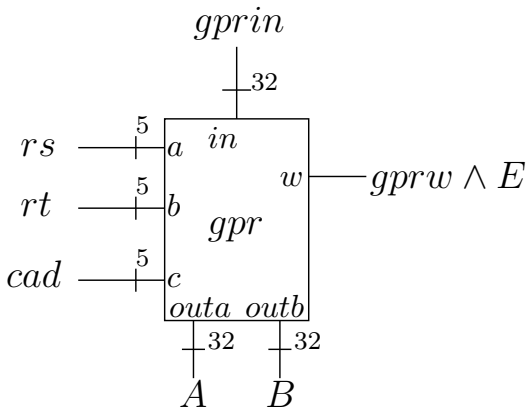


Figure 8.8: General purpose register file

One shows

$$bf(h) = bf(c)$$

in the same way. Bit $af[3](c)$ is a predicate, thus $af(h)$ is computed in the function decoder as a predicate and we get by Lemma 44

$$af[3](h) = af[3](c).$$

Because $i(c) = itype(c)$ is a predicate we get

$$i(h) = i(c)$$

directly from lemma 44. We summarize in

Lemma 48.

$$\begin{aligned} af(h) &= af(c) \\ i(h) &= i(c) \\ bf(h) &= bf(c) \end{aligned}$$

That finishes the bookkeeping of what the instruction decoder does.

8.3.7 Reading from General Purpose Registers

The general purpose register file $h.gpr$ of the hardware implementation as shown in Figure 8.8 is a 3 port GPR-RAM with two read ports and one write port. The a and b addresses of the file are connected to $rs(h)$ and $rt(h)$. For the data outputs $gprouta$ and $gproutb$ we introduce the shorthands A and B.

Then, we have

$$\begin{aligned} A(h) &= gprouta(h) \\ &= h.gpr(rs(h)) \quad \text{(construction)} \\ &= c.gpr(rs(h)) \quad \text{(Equation 8.1)} \\ &= c.gpr(rs(c)) \quad \text{(Lemma 45)} \end{aligned}$$

and, in an analogous way,

$$B(h) = c^i.gpr(rt(c)).$$

Thus, we have

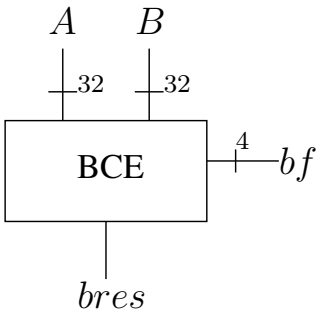


Figure 8.9: The branch condition evaluation unit and its operands

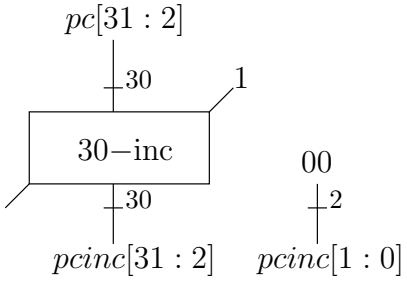


Figure 8.10: Incrementing an aligned PC with a 30-incrementer

Lemma 49 (Correctness of Values Read From GPR)**.**

$$\begin{aligned} A(h) &= c.gpr(rs(c)) \\ B(h) &= c^i.gpr(rt(c)) \end{aligned}$$

8.3.8 Next PC Environment

Branch Condition Evaluation Unit. The BCE-unit is wired as shown in Figure 8.9. By lemmas 49 and 44 as well as the correctness of the BCE implementation from Section 7.5 we have

$$\begin{aligned} bres(h) &= bceres(A(h), B(h), bf(h)) \quad \text{(construction)} \\ &= bceres(c.gpr(rs(c)), c.gpr(rt(c)), bf(c)) \quad \text{(Lemmas 49, ??)} \\ &= bres(c). \quad \text{(ISA definition)} \end{aligned}$$

Thus, we have

Lemma 50 (Branch Result Correctness)**.**

$$bres(h) = bres(c)$$

Incremented PC. The computation of an incremented PC as needed for the next PC environment as well as for the link instructions is shown in Figure 8.10. Because the PC can be assumed to be aligned[4] the use of a 30-incrementer suffices. Using the correctness of the incrementer from Section

[4]Otherwise a misalignment interrupt would be signalled.

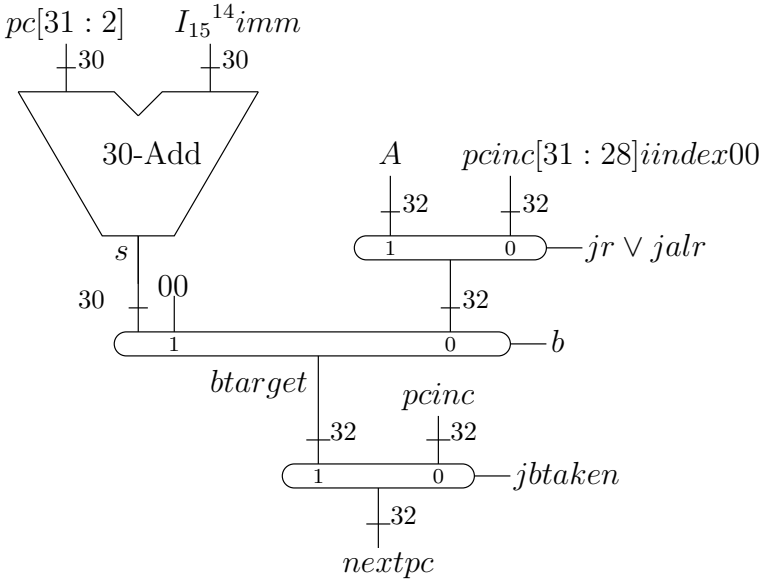


Figure 8.11: Next PC computation

7.1 we get

$$\begin{aligned} pcinc(h) &= (h.pc[31:2] +_{30} 1_{30})00 \\ &= (c.pc[31:2] +_{30} 1_{30})00 \quad \text{(Equation 8.1)} \\ &= c.pc[31:2]00 +_{32} 1_{30}00 \quad \text{(Lemma 20)} \\ &= c.pc +_{32} 4_{32}. \quad \text{(alignment)} \end{aligned}$$

Thus, we have

Lemma 51 (Incremented Program Counter Correctness)**.**

$$pcinc(h) = c.pc +_{32} 4_{32}$$

Next PC Computation. The circuit computing the next PC input, which was left open in Figure 8.4 when we treated instruction fetch, is shown in Figure 8.11.

Predicates $p \in \{jr, jalr, jump, b\}$ are computed in the instruction decoder. Thus, we have

$$p(h) = p(c)$$

by Lemma 44.

We compute *jbtaken* in the obvious way and conclude with Lemma 50

$$\begin{aligned} jbtaken(h) &= jump(h) \vee b(h) \wedge bres(h) \\ &= jump(c) \vee b(h) \wedge bres(c) \\ &= jbtaken(c). \end{aligned}$$

We have

$$\begin{aligned} A(h) &= c.gpr(rs(c)) \quad \text{(by Lemma 49)} \\ nextpc(h) &= c.pc +_{32} 4_{42} \quad \text{(by Lemma 51)} \\ imm(h)[15]^{14}imm(h)00 &= imm(c)[15]^{14}imm(c)00 \quad \text{(by Lemma 45)} \\ &= bdist(c). \end{aligned}$$

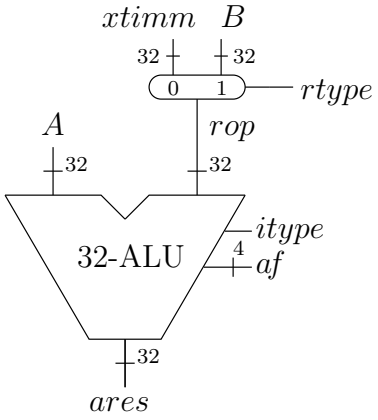


Figure 8.12: ALU environment

For the computation of the 30-bit adder we argue as in Lemma 51:

$$\begin{aligned} s(h)00 &= (h.pc[31:2] +_{30} imm(h)[15]^{14} imm(h))00 \\ &= (c.pc[31:2] +_{30} imm(c)[15]^{14} imm(c))00 \quad \text{(Lemma 45)} \\ &= c.pc[31:2]00 +_{32} imm(c)[15]^{14} imm(c)00 \quad \text{(Lemma 20)} \\ &= c.pc +_{32} bdist(c). \quad \text{(alignment)} \end{aligned}$$

We conclude

$$\begin{aligned} btarget(h) &= \begin{cases} c.pc +_{32} bdist(c) & b(c) \\ c.gpr(rs(c)) & jr(c) \vee jalr(c) \\ (c.pc +_{32} 4_{32})[31:28] iindex(c)00 & j(c) \vee jal(c) \end{cases} \\ &= btarget(c). \end{aligned}$$

Exploiting

$$reset(h) = 0$$

as well as the semantics of register updates we conclude

$$\begin{aligned} h'.pc &= nextpc(h) \\ &= \begin{cases} btarget(c) & jbtaken(c) \\ c.pc +_{32} 4_{32} \end{cases} \\ &= c'.pc. \end{aligned}$$

Thus, we have shown

Lemma 52 (Next Program Counter Correctness)**.**

$$h'.pc = c'.pc$$

This shows the first part of $c' \sim h'$.

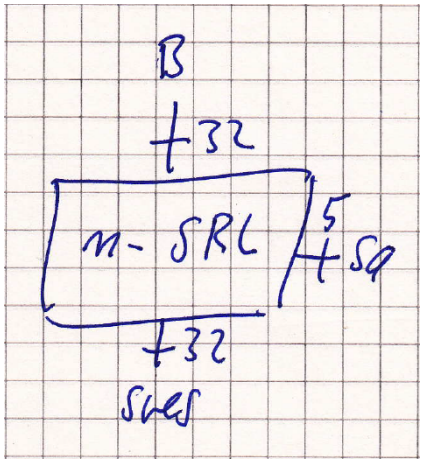


Figure 8.13: Shifter environment

8.3.9 ALU Environment

The ALU environment is shown in Figure 8.12. For the ALU's left operand we have

$$\begin{aligned} lop(h) &= A(h) \quad \text{(construction)} \\ &= c.gpr(rs(c)) \quad \text{(Lemma 49)} \\ &= lop(c). \end{aligned}$$

For the right operand follows with Lemmas 49, 47 and **??**

$$\begin{aligned} rop(h) &= \begin{cases} B(h) & rtype(h) \\ xtimm(h) & \text{otherwise} \end{cases} \\ &= \begin{cases} c.gpr(rt(c)) & rtype(c) \\ xtimm(c) & \text{otherwise} \end{cases} \\ &= rop(c). \end{aligned}$$

For the result $ares$ of the ALU we get

$$\begin{aligned} ares(h) &= alures(lop(h), rop(h), itype(h), af(h)) \quad \text{(Section 7.3)} \\ &= alures(lop(c), rop(c), itype(c), af(c)) \quad \text{(Lemmas 49, ??)} \\ &= ares(c). \end{aligned}$$

We summarize in

Lemma 53 (ALU Result Correctness)**.**

$$ares(h) = ares(c)$$

Note, that in contrast to previous lemmas the proof of this lemma is not just bookkeeping; it involves the not so trivial correctness of the ALU implementation from Section 7.3.

8.3.10 Shifter Environment

The shifter environment is shown in figure 8.13. For the shifters left operand we have

$$\begin{aligned} slop(h) &= B(h) \quad \text{(construction)} \\ &= c.gpr(rt(c)) \quad \text{(Lemma 49)} \\ &= slop(c). \end{aligned}$$

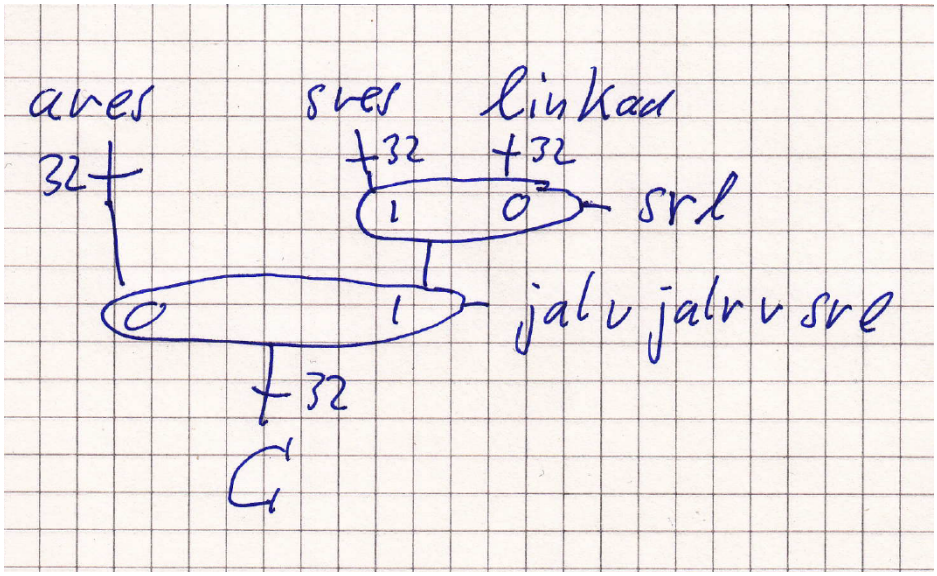


Figure 8.14: Collecting results into signal C

The shift distance is

$$sa(h) = sa(c)$$

by lemma 45. The result of the shifter with these operands is

$$\begin{aligned} sres(h) &= shres(slop(h), sa(h)) \quad \text{(Section 7.4)} \\ &= shres(slop(c), sa(c)) \\ &= sres(c). \end{aligned}$$

We summarize

Lemma 54.

$$sres(c) = sres(h)$$

8.3.11 Jump and Link

The value *linkad* that is saved in jump and link instructions is identical with the incremented PC *pcinc* from the next PC environment:

$$linkad(h) = pcinc(h) = h.pc +_{32} 4_{32} = linkad(c). \tag{8.2}$$

8.3.12 Collecting Results

Figure 8.14 shows two multiplexers collecting results *linkad*, *sres* and *ares* into an intermediate result C. Using Lemmas 53, 54 and **??** as well as Equation 8.2 we conclude

$$\begin{aligned} C(h) &= \begin{cases} linkad(h) & jal(h) \vee jalr(h) \\ sres(h) & srl(h) \\ ares(h) & \text{otherwise} \end{cases} \\ &= C(). \end{aligned}$$

Thus, we have

Lemma 55 (C Result Correctness)**.**

$$C(h) = C(c)$$

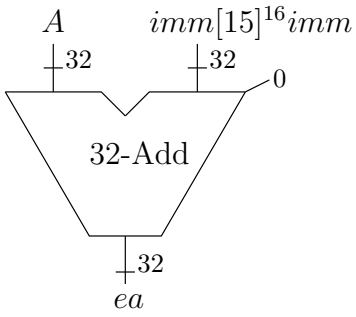


Figure 8.15: Effective address computation

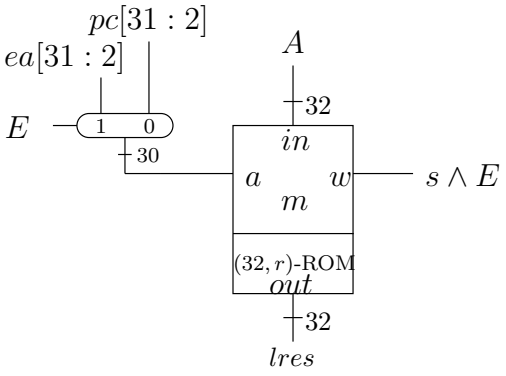


Figure 8.16: Memory environment

8.3.13 Effective Address

The effective address computation is shown in Figure 8.15. We have

$$\begin{aligned} ea(h) &= A(h) +_{32} imm(h)[15]^{16}imm(h) \quad \text{(Section 7.1)} \\ &= c.gpr(rs(c)) +_{32} sxtimm(c) \quad \text{(lemmas 49 and 45)} \\ &= ea(c). \end{aligned}$$

Thus, we have

Lemma 56 (Effective Address Correctness)**.**

$$ea(h) = ea(c)$$

8.3.14 Memory Environment

We implement only word accesses. Figure 8.16 shows the inputs to the ROM-RAM used as memory.

Here, one has to show

Lemma 57 (Memory Implementation Correctness)**.**

$$lres(h) = lres(c)$$

and

$$h'.m \sim_M c'.m$$

Proof. The proof of the first statement is completely analogous to the analysis of instruction fetch using $ea(h) = ea(c)$ and concluding $ma(h) = ea(h)[32 : 2]$ from $h.E = 1$. For the second statement we have to consider the obvious case split

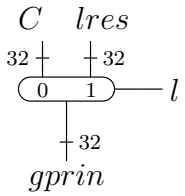


Figure 8.17: Computing the data input of the GPR

- $s(h) = 0$, i.e. no store is performed. From lemma 44 we get $s(c) = 0$, hence $mw(h) = 0$. Using hypothesis $c \sim_M h$ we conclude for all word addresses a

$$\begin{aligned} h'.m(a) &= h.m(a) \\ &= c.m_4(a00) \\ &= c'.m_4(a00) \end{aligned}$$

- $s(h) = 1$ i.e. a store with address $ea(h)[32:2]$ is performed. From lemma 44 we get $s(c) = 0$, hence $mw(h) = 1$. For word addresses $a \neq ea(h)[32:2]$ one argues

$$h'(a) = c'.m_4(a00)$$

as above. For $a = ea(h)$ we use lemma 49 to conclude

$$min(h) = c.gpr(rs(c))$$

and conclude

$$\begin{aligned} h'.m(ea(h)[32:2]) &= c.gpr(rs(c)) \\ &= c'_4.m \quad \text{(software condition about } ea \text{ of stores)} \\ &= c'_4.m(ea(c)[32:2]00) \quad \text{(alignament)} \\ &= c'_4.m(ea(h)00) \end{aligned}$$

Note that in this proof we argued that *both* the ISA memory and the hardware memory are updated at the locations concerned. This only works if the effective address does not lie in ROM.

□

8.3.15 Writing to the General Purpose Register File

Figure 8.17 shows a last multiplexer connecting the data input of the general purpose register file with intermediate result C and the result $lres$ coming from the memory.

Using lemmas 55 and **??** we conclude

$$\begin{aligned} gprin(h) &= \begin{cases} lres(h) & l(h) \\ C(h) & \text{otherwise} \end{cases} \\ &= \begin{cases} lres(c) & l(c) \\ C(c) & \text{otherwise} \end{cases} \\ &= gprin(c). \end{aligned}$$

Using $h.E01$ and **??** we get

$$gprw(h) = gprw(c)$$

With hypothesis $c \sim h$ we conclude for the general purpose register file:

$$\begin{aligned} h'.gpr(x) &= \begin{cases} gprin(h) & gprw(h) \wedge x = cad(h) \\ h.gpr(x) & \text{otherwise} \end{cases} \\ &= \begin{cases} gprin(c) & gprw(c) \wedge x = cad(c) \\ c.gpr(x) & \text{otherwise} \end{cases} \\ &= c.gpr(x). \end{aligned}$$

This concludes the proof of lemma 43 and also the induction step of the correctness proof of the entire (simple) processor.

8.4 Example Programs

Before we present example programs, we have to say a few words about the representation of constants in assembly language. In general, register numbers and immediate constants are represented in decimal notation. Conversion to two's complement representation is done by the assembler. If we want to specify an immediate constant $i[15:0)$ in binary representation we write it as

$$0bi[15:0]$$

We put comments in brackets. Sometimes we number lines. These numbers are comments too; they just serve to compute jump distances

8.4.1 Simple MIPS Programs

Initializing register 0 with all zeros This is done by XORing the register with itself

```
xor 0 0 0
```

We abbreviate this program with

$$gpr(0) = 0$$

Storing a 32 bit constant in a register . Let $i[31:0] \in \mathbb{B}^{32}$. The following program loads this constant in register k in two instruction. First it stores the upper half and then the zero extended lower half is ORed with this

```
lhi k 0bi[31:16]
ori k k 0bi[15:0]
```

We abbreviate this program with

$$loadc(k, 0bi[31:0])$$

Computing sign and absolute value of an integer . Assume a two's complement number is stored in register $gpr(i)$. We want to store sign and absolute value of this number in registers j and k. For the sign we simply compare $gpr(i)$ with zero

```
0: gpr(0)=0
1: slt j i 0
```

For the absolute value we invert and increment in case the number is negative. We invert by xoring with the mask 1^{32} which is obtaind by a nor of the mask 0^{32} with itself

```
2: blez j 4
3: nor k 0 0
4: xor k k i
5: addi k k 1
```

8.4.2 Software Multiplication

The following program takes initially (in configuration c^0) binary numbers a and b from registers $gpr(i)$ and $gpr(j)$ as inputs.

$$\begin{aligned} c^0.gpr(i) &= a \\ c^0.gpr(j) &= b \end{aligned}$$

It should compute finally (in some configuration c^T) the product mod 2^{32} of these numbers in some register k. Thus we want to show

$$\langle c^T.gpr(k)\rangle = (\langle a\rangle \cdot \langle b\rangle \bmod 2^{32})$$

We start by setting register 0 to 0^{32}

```
0: xor 0 0 0 (gpr(0) = 0)
```

We use four auxiliary registers $gpr(r)$ for $r \in [24:27]$ that we abbreviate in comments as

$$\begin{aligned} x &= gpr(24) \\ y &= gpr(25) \\ z &= gpr(26) \\ u &= gpr(27) \end{aligned}$$

We initialize x with $0^{31}1$ and y with a and u with 0^{32}. Then we AND b and x together

```
0: addi 24 0 1   (x=1)
1: addi 26 i 0   (z=a)
2: addi 27 0 0   (u=0)
3: and 25 j 24   (y = b AND x)
```

Afterwards we have

$$y \neq 0 \leftrightarrow b[0] = 1$$

We add $z = a$ to u if b[0]=1

```
4: bne 25 0 2      (skip next instruction if b[0]=0)
5: add 27 27 26    (u=u+z)
```

Afterwards we have

$$\begin{aligned} x &= 0^{31}1 \\ z &= a \\ \langle\rangle &= \langle a\rangle b[0] \end{aligned}$$

The next lines will contain a loop. After the loop has been processed i times the following will hold

$$\begin{aligned} x &= \begin{cases} 0^{31-i}10^i & i < 31 \\ 0^{32} & i = 31 \end{cases} \\ \langle z\rangle &= \langle a\rangle \cdot 2^i \bmod 2^{32} \\ \langle u\rangle &= \langle a\rangle \cdot \langle b[i:0]\rangle \end{aligned}$$

As we have seen this is true initially for $i = 0$, and we program the loop just to maintain this 6: beq 24 0 7 (jump over next 6 instructions if 32 iterations are over)

```
7: add 24 24 24 (double x mod 2^32)
8: add 26 26 26 (double a mod 2^32)
9: and 25 j 24  (y = b AND x)
```

we have at this point in iteration i:

$$y \neq 0 \leftrightarrow b[i] = 1$$

We add z to u if b[i]=1 and go back 6 instructions, i.e. to the start of the loop

```
10: bne 25 0 2      (skip next instruction if b[i]=0)
11: add 27 27 26    (u=u+z)
12: beq 0 0 -6      (go back 6 instructions to start of )
```

After 32 iterations we have the desired product in u. In the end we copy the result to from u register j

```
13: addi j 27 0   (gpr(j) = u)
```

We abbreviate the above program with

$$mul(i, j, k)$$

Note that the program can be used literally for integer multiplication tmod 2^{32} too. The argument is known from the correctness of arithmetic units

By lemmas 22 and 8 we have in the end

$$\begin{aligned} [a] \cdot [b] &\equiv \langle a \rangle \cdot \langle b \rangle \bmod 2^{32} \\ &= \langle u \rangle \bmod 2^{32} \\ &= [u] \bmod 2^{32} \end{aligned}$$

As $[u] \in T_{32}$ we get by lemma 11

$$[u] = [a] \cdot [b] \quad tmod \quad 2^{32}$$

8.4.3 School Method for Integer Division

For integers a and b with $b \neq 0$ the integer division of a by b can be defined as

$$a/b = max\{C \in \mathbb{N} : b \cdot C \leq a\}$$

If $a < b$, then $a/b = 0$ and we are done. Otherwise we transfer the school algorithm for long division of decimal numbers to binary numbers. A single step of the algorithm is justified by

Lemma 58. *For $a > b$ let*

$$t(a, b) = max\{k \in \mathbb{N} : 2^k \cdot b \leq a$$

then

$$a/b = 2^{t(a,b)} + (a - 2^{t(a,b)} \cdot b)/b$$

Proof. a/b has a binary representation $c[m:0] \in \mathbb{B}^{m+1}$ without leading zeros.

$$a/b = \langle 1c[m-1:0]\rangle$$

i.e.

$$\langle 1c[m-1:0]\rangle \cdot b \leq a$$

and

$$\langle (1c[m-1:0]\rangle + 1) \cdot b > a$$

We show

$$\begin{aligned} m &= t(A,B) \\ \langle c[m-1:0]\rangle &= (a - 2^{t(a,b)} \cdot b)/b \end{aligned}$$

We prove the two equations separately.

- We have

  $$2^m \cdot b \leq \langle 1c[m-1:0]\rangle \cdot b \leq a$$

  Hence

  $$m \leq t(a,b)$$

  Assuming $m < t(a,b)$ gives the contradiction

  $$\langle (1c[m-1:0]\rangle + 1) \cdot b \leq 2^{m+1} \cdot b \leq a$$

- We have

  $$\begin{aligned} (2^m + \langle c[m-1:0]\rangle) \cdot b &\leq a \\ \langle c[m-1:0]\rangle \cdot b &\leq a - 2^m \cdot b \\ \langle c[m-1:0]\rangle &\leq (a - 2^m \cdot b)/b \end{aligned}$$

  Assuming

  $$(\langle c[m-1:0]\rangle + 1) \cdot b \leq a - 2^m \cdot b$$

  Leads to the contradiction

  $$(\langle 1c[m-1:0]\rangle + 1) \cdot b = (2^m + \langle c[m-1:0]\rangle + 1) \cdot b = a$$

□

Observe that with the notation of the above proof we have

Lemma 59.

$$a - 2^m \cdot b < 2^m \cdot b$$

Proof. The assumption

$$a - 2^m \cdot b \geq 2^m \cdot b$$

leads to the contradiction

$$a \geq 2^{m+1} \cdot b = 2^{t(a,b)+1} \cdot b$$

□

For $a > b$ we obtain the school method for iterative binary division in the obvious way

- $$A(0) = a$$

- For $A(i) \geq b$ we define

$$\begin{aligned} t(i) &= t(A(i), b) \\ A(i+1) &= A(i) - 2^{t(i)} \cdot b \end{aligned}$$

From lemma 59 we get

$$A(i+1) < 2^{t(i)} \cdot b$$

Hence

$$\begin{aligned} t(i+1) &< t(i) \\ t(i) &\leq t(0) - i \\ A(i) &< 2^{t(0)-i} \cdot b \end{aligned}$$

and the algorithm terminates after u steps with

$$u = max\{i : A(i) \geq b\}$$

An easy induction on i with lemma 58 gives

$$a/b = \sum_{j=0}^{i} 2^{t(i)} + A(i+1)/b$$

Thus we have

Lemma 60.

$$a/b = \sum_{i=0}^{u} 2^{t(i)}$$

8.4.4 Implementing Integer Division

The following algorithm takes initially (in configuration c^0) binary numbers p and q from registers $gpr(i)$ and $gpr(j)$ and should finally (in some configuration c^R) compute the result of the integer division in some register $gpr(k)$.

$$\begin{aligned} c^0.gpr(i) &= p \\ c^0.gpr(j) &= q \\ c^R.gpr(k) &= \langle p \rangle / \langle q \rangle \end{aligned}$$

Obviously we assume $b \neq 0^{32}$. In order to avoid a tedious case distinction in the case $t(0) = 31$ we also assume that the leading bit of a is zero.

$$q \neq 0^{32} \wedge p[31] = 0$$

We abbreviate

$$a = \langle p \rangle \quad \text{and} \quad b = \langle q \rangle$$

We use auxiliary registers that we abbreviate

$$\begin{aligned} A &= gpr(23) \\ B &= gpr(24) \\ X &= gpor(25) \\ C &= gpr(26) \\ U &= gpr(27) \end{aligned}$$

We initialize $gpr(0)$ and the result register $gpr(k)$ with zero. With an initial test we handle the case $a < b$ and simply jump to the end of the routine

```
0: gpr(0)=0
1: add k 0 0      (gpr(k) = 0)
2: sltu 23 i j    (A = (a < b))
3: bgez 23 17       (quit, i.e. goto line 20 if a<b)
```

We initialize

```
4: add 23 i 0       (A = p)
5: add 24 j 0       (B = q)
6: addi 25 0 1      (X = 0...01)
7: add 26 0 0       (C = 0)
```

and have after execution of this code

$$\begin{aligned} \langle A\rangle &= a \\ \langle X &= b \cdot 2^0 \\ X[j] = 0 &\leftrightarrow j = 0 \\ \langle C\rangle &= 0 \end{aligned}$$

Next we successively double $\langle X\rangle$ and $\langle B\rangle$ until $\langle B\rangle > \langle A\rangle$. This will succeed after some number m of iterations because $q[31:0]$ contains a one and p has a leading zero.

```
8: add 25 25 25      (X = X + X)
9: add 24 24 24      (B = B + B)
10: sltu 27 23 24    ( U = (A < B))
11: blez 27 -3       (repeat if B <= A)
```

This loop ends after m iterations for some m and we have

$$\begin{aligned} \langle A\rangle &= a \\ \langle B\rangle &= 2^m \cdot b \\ \langle X[j] = 1 &\leftrightarrow j = m \\ 2^m \cdot b &> a \\ m &> t(0) \end{aligned}$$

We write the main loop such that after i iterations the following is satisfied

$$\begin{aligned} \langle A\rangle &= A(i) \\ \langle B\rangle &= 2^{t(i-1} \cdot b \\ X[j] = 1 &\leftrightarrow j = t(i-1) \\ \langle C\rangle &= \sum_{j=0}^{i-1} 2^{t(i)} \end{aligned}$$

This obviously holds for $i = 0$ and is maintained in iteration $i + 1$ by the following code.

```
12: srl 25 25 1       (X = srl(X,1))
13: srl 24 24 1       (B = srl(B,1))
14: sltu 27 23 24     (U = (A < B))
15: blez 27 -3        (repeat if A >= B)
16: or 26 26 x        ( C = C OR X)
17: sub 23 23 24      (A = A - B)
18: sltu 27 23 j      (U = (A < b))
19: blez 27 -7        (repeat outer loop if A >=b)
```

In the inner loop (lines 12 - 15) B and X are shifted to the right until

$$\langle B \rangle < \langle A \rangle = A(i)$$

. At this point we have

$$X[j] = 1 \leftrightarrow j = t(i)$$

We OR X into C which gives a new value

$$\begin{aligned} \langle C \rangle &= \sum_{j=0}^{i-1} 2^{t(i)} + 2^{t(i)} \\ &= \sum_{j=0}^{i} 2^{t(i)} \end{aligned}$$

Finally we subtract $\langle B \rangle$ from $\langle A \rangle$ obtaining a new value A which satisfies

$$\begin{aligned} \langle A \rangle &= A(i) - 2^{t(i)} \cdot b \\ &= A(i+1) \end{aligned}$$

Chapter 9

Context Free Grammars

9.1 Introduction to Context Free Grammars

9.1.1 Syntax of Context Free Grammars

A context free grammar G has the following components

- a finite set of symbols $G.T$ called alphabet of terminal symbols
- a finite set of symbols $G.N$ called the alphabet of non terminal symbols. Symbols cannot be simultaneously terminal and nonterminal:

$$G.T \cap G.N = \emptyset$$

- a start symbol $S \in G.N$
- a finite set $P \subset G.N \times (G.N \cup G.T)^*$ of productions

If $(n, w_0 \dots w_{k-1})$ is a production of grammar G, i.e. $(n, w_0 \dots w_{k-1}) \in P$ we say that the string $w_0 \dots w_{k-1}$ is directly derived in G from the nonterminal symbol n, and we write

$$n \rightarrow_G w_0 \dots w_{k-1}$$

If there are several strings $w^1, \dots, w^s$ that are directly derived in G from n and if $| \notin G.N \cup G.T$, then we write

$$n \rightarrow_G w^1 | \dots | w^s$$

If it is clear which grammar is meant we abbreviate

$$\begin{aligned} T &= G.T \\ N &= G.N \\ S &= G.S \\ P &= G.P \\ \rightarrow &= \rightarrow_G \end{aligned}$$

Before we present an example a word on notation. In mathematics there is a constant shortage of symbols. One uses capital and small letters of the Latin and Greek alphabets. It is not enough. One borrows - in set theory - from the Hebrew alphabet. One introduces funny letters like $\mathbb{B}$ and $\mathbb{N}$. The shortage remains. Modification of symbols with accents and tildes is a more radical step: using a symbol naked, with an accent or with a tilde immediately triples the number of available symbols. In computer science one has taken the following brute force measure to generate arbitrarily many

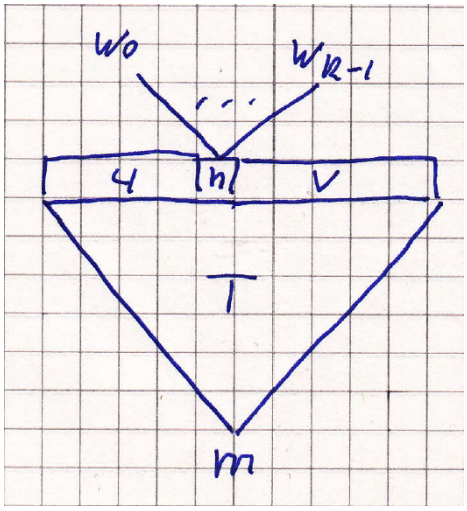


Figure 9.1: Extending derivation tree T by application of production $n \to w$

symbols: take a non empty alphabet A not containing the symbols $\langle$ and $\rangle$. There are arbitrarily many strings $w \in A^*$. For each such w treat $\langle w \rangle$ as a new symbol. If we write down a string of such symbols the pointed brackets permit to decide where symbols begin and end.

As an introductory example we consider the following grammar:

$$\begin{aligned} T &= \{0, 1, X\} \\ N &= \{V, C, \langle CS \rangle\} \\ S &= V \\ B &\to 0|1 \\ \langle BS \rangle &\to B|B\langle BS \rangle \\ V &\to X\langle BS \rangle|0|1 \end{aligned}$$

9.1.2 Quick and Dirty Introduction to Derivation Trees

Let $G = (T, N, P, S)$ be a context free grammar. Intuitively and with the help of drawings it is extremely easy to explain, what is a derivation tree T for grammar G. For nonterminals $n \in N$ and strings $w \in (N \cup T)^*$ we generate the derivation trees T with root n and border word w by the following rules, which are illustrated in figure 9.1.

1. a single nonterminal n is a derivation tree with root n and border word n.

2. if T is a derivation tree with root m border word unv and $n \to w \in P$ is a production with $w = w_0 \dots w_{k-1}$, then the tree in figure x a) is a derivation tree with root m and border word uwv.

3. all derivation trees can be generated by finitely many applications of the above 2 rules.

In the example grammar above we can construct a derivation tree with root V and border word $X01$ by 6 applications of the above rules as illustrated in figure 9.2.

We can argue that derivation trees can be composed:

Lemma 61. *If T_1 is a derivation tree with root n and border word unv and T_2 is a derivation tree with root n and border word w, then the tree T from figure x is a derivation tree with root n and border word uwv.*

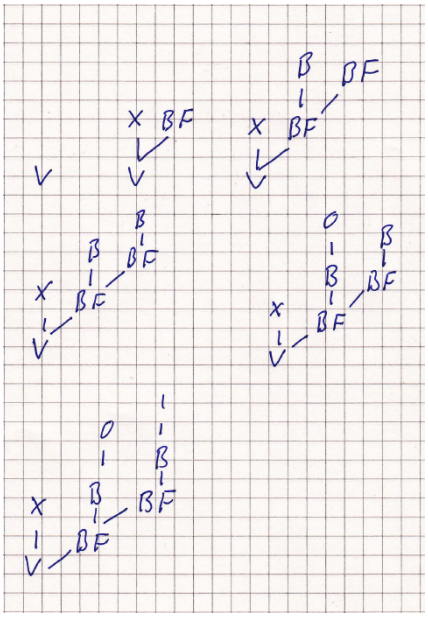


Figure 9.2: Generating a derivation tree with root V and border word $X01$

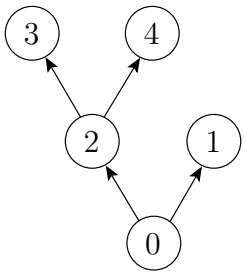


Figure 9.3: Drawing the rooted tree from figure 3.6 in a different way.

Proof. generate tree T by first applying the rules for generating T_1 and then using the rules for generating T_2 □

We also can define T_2 to be a subtree of T if T can be generated by the composition of T_2 with some derivation tree T_1 as in lemma 61. This concept of derivation trees goes a long way. Problems arise however, for instance if we try to make precise, when two such trees are equal. As we have done 'definitions' by picture we would have to define: two trees are equal if they have the same drawing. This is intuitively quite clear[1] but not a mathematical definition. Much more serious problems arise when we try to argue about the behavior of algorithms, which traverse and process derivation trees. When we try to argue, what happens, when the algorithm visits a certain node, we have no language to do so, simply because nodes have no names. Compilers, however, happen to work in exactly this way. Thus we give in the next subsections a clean definition of derivation trees and relate it to the basic graph theory from section 3.4.

9.1.3 Tree Regions

The graph theoretic definition of rooted trees from subsection 3.4.3 does not specify the order of sons. It does not distinguish between the graphs drawn in figure 3.3 and in figure 9.3. In what follows however we cannot ignore that order. Therefore we will formalize trees such that

- for each node u with outdegree k the sons of u are numbered from 0 to $k-1$

[1] A picture says more than a thousand words

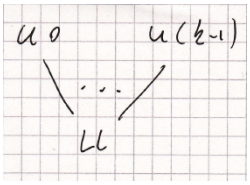


Figure 9.4: Standardized names of nodes specify the order of sons

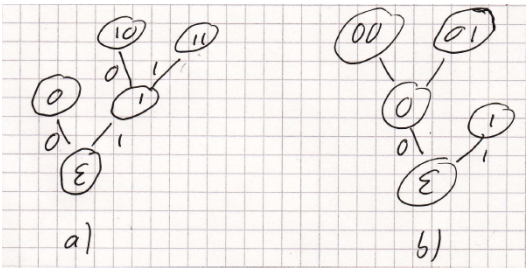


Figure 9.5: Names of nodes correspond to edge labels on the paths from the root to the nodes

- nodes have standardized names $u \in \mathbb{N}^*$. The root has name $r = \epsilon$ and the sons of a node u with outdegree k are called $u \circ 0, \ldots, u \circ k-1$. When we draw the trees node $u \circ i$ is drawn to the left of $u \circ (i+1)$ (see figure 9.4).

Actually this naming is motivated by graph algorithms working on the graphs starting at the root. If one labels the edges of a node with k sons from 0 to $k-1$ then the name v of a node now is simply the sequence of edge labels encountered on the path from the root to v. For the graphs in figures 3.3 and 9.3 this is illustrated in figures 9.5 a) and b). For a formal definition of tree regions we define from sets A of sequences of natural numbers

$$A \subset \mathbb{N}^*$$

graphs $G(A)$ in the following way

- the set of nodes of $G(A)$ is A

$$G(A).V = A$$

- for $u, v \in A$ there is an edge from u to v if $v = u \circ i$ for an $i \in \mathbb{N}$

$$(u, v) \in G(A).E \leftrightarrow \exists i \in \mathbb{N} : v = u \circ i$$

For nodes $v \in A$ we can characterize their indegree and outdegree in the graph $G(A)$ by

Lemma 62.

$$\begin{aligned} outdeg(v, G(A)) &= \#\{i : i \in \mathbb{N}, v \circ i \in A\} \\ indeg(\varepsilon, G(A)) &= 0 \\ indeg(v \circ i, G(A)) &= \begin{cases} 1 & v \in A \\ 0 & v \notin A \end{cases} \end{aligned}$$

We define again a node v to be a source of A if $indeg(v, G(A)) = 0$, i.e. if there is no node $u \in A$ and no $i \in \mathbb{N}$ such that $v = u \circ i$. We define u to be a sink of A if $outdeg(u, G(A)) = 0$, i.e. if for no $i \in \mathbb{N}$ we have $v = u \circ i \in A$.

Now we define tree regions as the subsets $A \subset \mathbb{N}^*$ such that the graphs $G(A)$ are rooted trees with root ϵ. Formally

1. $A = \{\varepsilon\}$ is a tree region

2. if A is a tree region, v is a sink of A and $k \in \mathbb{N}$ then

$$A' = A \cup \{(v \circ 0), \ldots (v \circ k-1)\}$$

   is a tree region

3. all tree regions can be generated by finitely many applications of these rules

A trivial argument shows

Lemma 63. *If A is a tree region, then $G(A)$ is a rooted tree.*

The father $father(u)$ of a node $u = u[1:n]$ in a tree region is defined as

$$father(u[1:n]) = u[1:n-1]$$

Formally the tree region of figure 9.5 a) is specified by

$$A = \{\varepsilon, 0, 1, 10, 11\}$$

and the tree region of figure 9.5 b) is specified by

$$A = \{\varepsilon, 0, 1, 00, 01\}$$

9.1.4 Clean definition of derivation trees

A derivation tree T for grammar G has the following components

- a tree region $T.A$

- a labeling $\ell : A \to G.N \cup G.T$ of the elements of $T.A$ by symbols of the grammar satisfying: if $outdeg(u, G(T.A)) = k$, and $\ell(u) = n$ and for all i we have $\ell(u \circ i) = w_i$ then

$$n \to_G w_0 \ldots w_{k-1}$$

  i.e. the sequence of the labels of the sons of u is directly derived in G from the label of u. (see figure 9.6 a)

There are three obvious ways to draw derivation trees T:

- For each node $u \in T.A$ in the drawing, we write down $u : \ell(u)$, i.e. the name and its label as in figure 9.6 a)

- we can omit the names of the node as shown in figure 9.6 b) and only draw edge label

- we can omit the edge labels as shown in figure 9.6. This is justified as edges startig at the same node are labeled from left to right starting with 0 . As long as the entire tree is drawn the names of the nodes can easily be reconstructed from the drawing. Also we know, that node labels in derivation trees are single symbols. If such a symbol happens to be an artificially generated symbol of the form $< w >$ we will take the freedom to omit the pointed brackets.

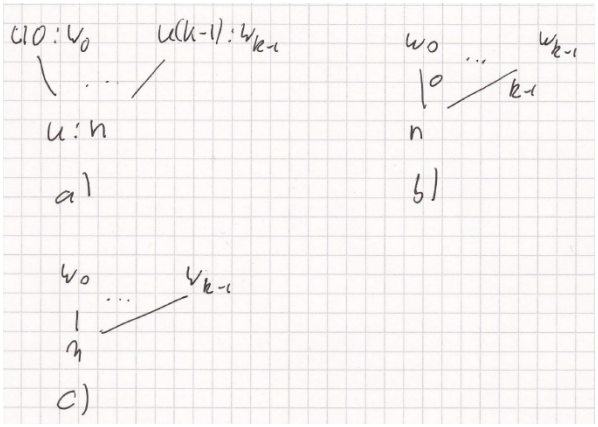


Figure 9.6: In derivation trees, nodes and their sons are labeled by productions of the grammar. Here $n \to w_1 \dots w_k$ must hold.

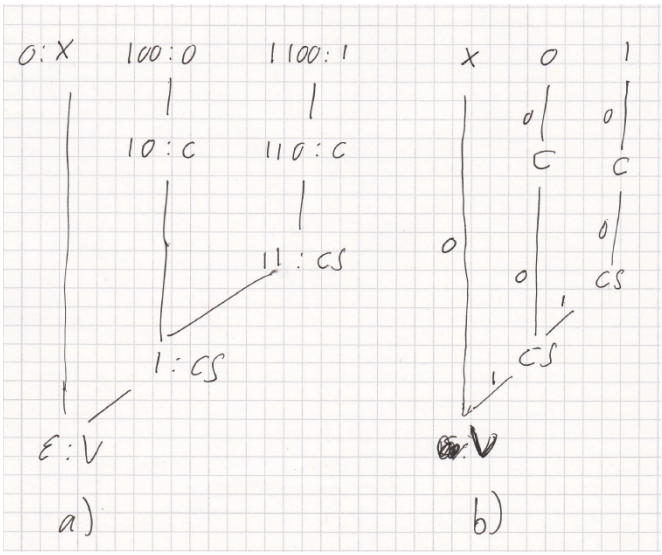


Figure 9.7: A derivation tree for the grammar of the introductory example. We have written CF instead of $\langle CF \rangle$. TODO: change C to B

A derivation tree for the grammar of the introductory example is shown in figure 9.7 a) with node names and in figure 9.7 b) with edge labels.

Let T be a derivation tree and $u \in T.A$ be a node in T. The border word $bw(u, T)$ is obtained by concatenating the labels of all leaves reachable from u from left to right. Formally

$$bw(u,T) = \begin{cases} T.\ell(u) & \text{u is a leaf of G(T.A)} \\ bw(u \circ 1, T) \circ \dots bw(u \circ k, T) & outdegree(u, G(T.A)) = k \end{cases}$$

If it is clear which tree is meant we drop the T.

In the derivation tree T of figure 9.7 we have for instance

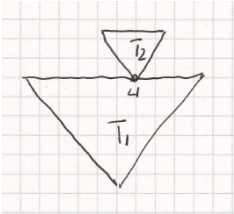


Figure 9.8: Composition of trees T_1 and T_2 at leaf u of T_1

$$\begin{aligned} bw(\epsilon) &= bw(0) \circ bw(1) \\ &= \ell(0) \circ bw(10) \circ bw(11) \\ &= X \circ bw(100) \circ bw(110) \\ &= X \circ \ell(100) \circ bw(1100) \\ &= X0 \circ \ell(1100) \\ &= X01 \end{aligned}$$

With the new precise definition of derivation trees we redefine relation $n \to_G w$. We say that a word w is derivable from n in G and write $n \to_G^* w$, if there is a derivation tree T for G with border word w whose root is labeled with n.

$$n \to_G^* w \leftrightarrow \exists T : T.\ell(\epsilon) = n \wedge bw(\epsilon, T) = w$$

9.1.5 Composition and Decomposition of Derivation Trees

The following easy theory permits now to argue about the composition and the decomposition of derivation trees. For the composition, let T_1 and T_2 be derivation trees for a common grammar G. Let u be a leaf of T_1 and assume that the labels of u in T_1 and the root ε in T_2 are identical:

$$T_1.\ell(u) = T_2.\ell(\varepsilon) \tag{9.1}$$

As illustrated in figure 9.8 we then we can compose T_1 and T_2 into a new tree $T = comp(T_1, T_2, u)$ in the following way. Nodes of T are the nodes of T_1 as well as the nodes of T_2 extended by prefix u

$$T.A = T_1.A \cup \{u\} \circ T_2.A$$

Labels of nodes in T are imported in the obvious way

$$T.\ell(v) = \begin{cases} T_1.\ell(v) & v \in T_1.A \\ T_2.\ell(x) & v = u \circ x \end{cases}$$

For $x = \varepsilon$ this is well defined by equation 9.1. An easy exercise shows

Lemma 64. *$T = comp(T_1, T_2, u)$ is a derivation tree*

For the decomposition let T be a derivation tree and $u \in T : A$ be a node of the tree. As illustrated in figure 9.9 we decompose T into the subtree $T_2 = sub(T, u)$ of T with root u and the tree $T_1 = rem(T, u)$ remaining from T if T_2 is removed. Formally

$$\begin{aligned} T_2.A &= \{v : u \circ v \in T : A\} \\ T_1.A &= (T.A \setminus T_2.A) \cup \{u\} \\ T_2.\ell(v) &= T.\ell(u \circ v) \\ T_1.\ell(v) &= T.\ell(v) \end{aligned}$$

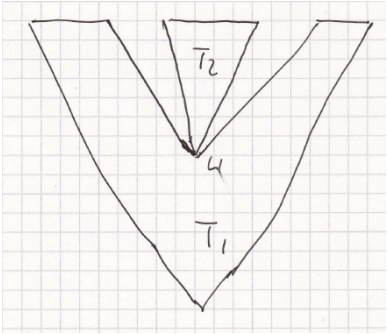


Figure 9.9: Decomposing a derivation tree into T_1 and T_2 at node u

An easy exercise shows

Lemma 65. *$T_1 = rem(T, u)$ and $T_2 = sub(T, u)$ ar both derivation trees*

Another easy exercise shows that if we decompose a tree T at node u and then compose the trees $T_1 = rem(T, u)$ and $T_2 = sub(T, u)$ again at u we get back - words of wisdom - the original tree.$T_1 = rem(T, u)$ and $T_2 = sub(T, u)$

Lemma 66.

$$T = comp(rem(T, u), sub(T, u), u)$$

Figures 9.8 and 9.9 differ only in the way we draw T_1. In figure 9.8 we stress that u it is a leaf of T_1. In figure 9.9 (where u is also a leaf of T_1) we stress that it is some node in the tree T.

9.1.6 Generated Languages

We say that a word w is derivable from n in G and write $n \to_G^* w$ if there is a derivation tree T for G with root n and border word w.

The language generated by nonterminal n in grammar G consists of all words in the terminal alphabet that are derivable from n in G. It is abbreviated with $L_G(n)$

$$L_G(n) = \{w : n \to_G^* w\} \cap G.T^*$$

An easy exercise shows for the grammar G of the introductory example

$$\begin{aligned} L_G(\langle CS \rangle) &= \mathbb{B}^+ \\ L_G(V) &= X\mathbb{B}^+ \cup \mathbb{B} \end{aligned}$$

Observe that we could drop the brackets of symbol $\langle CF \rangle$ because the arguments of function L_G are single symbols. The language generated by grammar G is the language $L_G(G.S)$ generated by G from the start symbol. A language is called context free if it is generated by some context free grammar.

9.2 Grammars for Expressions

9.2.1 Syntax of Boolean Expressions

In the hardware chapters we have introduced Boolean expressions by analogy to arithmetic expressions, i.e. we have not given a precise definition. We will now fill this gap by presenting a context

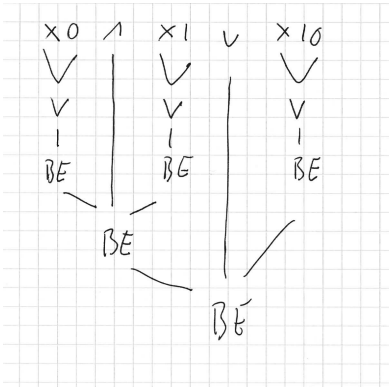


Figure 9.10: A derivation tree reflecting the usual priority rules

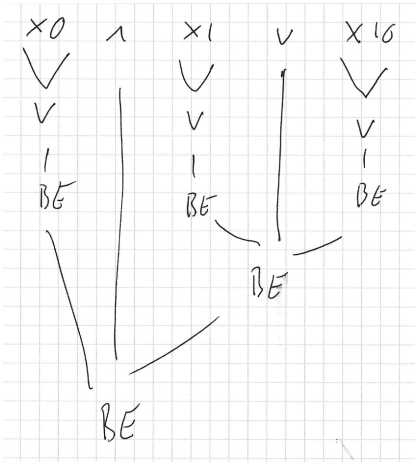


Figure 9.11: A derivation tree not reflecting the usual priority rules

free grammar that generates these expressions. We will consider variables in $X\mathbb{B}^+$ and constants in $\mathbb{B}$, and we use the productions of the grammar from the introductory example to produce such variables or constants from symbol V. Boolean expressions are derived from symbol $\langle BE\rangle$ by the productions

$$\begin{aligned}\langle BE\rangle \;\;\rightarrow\;\; & \langle BE\rangle \wedge \langle BE\rangle | \langle BE\rangle \vee \langle BE\rangle | \langle BE\rangle \oplus \langle BE\rangle \\ & | / \langle BE\rangle | (\langle BE\rangle)\end{aligned}$$

We make $\langle BE\rangle$ the start symbol of the extended grammar G. A derivation tree for the expression $X0 \wedge X1 \vee X10$ in G is shown – without the details for the derivation of the variables from symbol $\langle VC\rangle$ – in figure 9.10.

If we were to evaluate expressions as suggested by the shape of the tree, this tree would nicely reflect the usual priority among the Boolean operators, where $\wedge$ binds stronger than $\vee$.

However, the simple grammar G also permits a second derivation tree of the same Boolean expression, which is shown in figure 9.11. Although grammar G nicely represents the syntax of expressions, it is obviously not helpful for the definition of expression evaluation. For this purpose we need grammars G generating expressions with two additional properties:

- the grammars are unambiguous in the sense that for every word w in the language generated by G there is exactly one derivation tree for G of w from the start symbol.
- this unique derivation tree should reflect the usual priorities between operators

In the next subsection we exhibit such a grammar for arithmetic expressions.

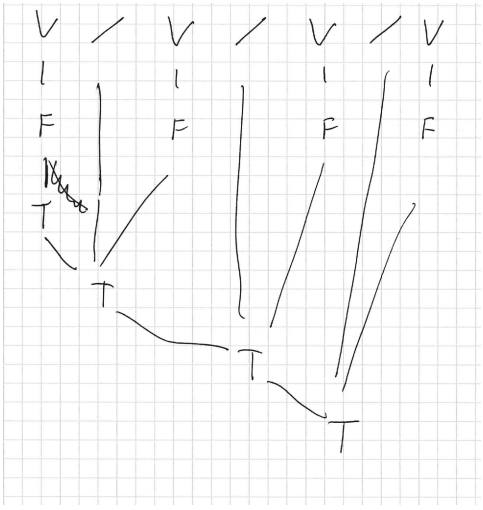


Figure 9.12: The grammar determines the order in which division signs are applied

9.2.2 Grammar for Arithmetic Expressions with Priorities

Evaluation of arithmetic expressions is more interesting meets the eye. People tend to believe that they have learnt expression evaluation at high school. Readers feeling this way are encouraged to solve the following

Exercise evaluate $2/2/2/2$ and $2 - - - -2$. We give two hints: i) evaluate first $2 - 2 - 2 - 2$. This gives you an idea about the order of operators. Then evaluate $2 - (-2)$. You will recognize, that there are two minus signs: a unary operator $-_1$ and a binary operator $-_2$. So you are really evaluating $2 -_2 (-_1 2)$. As in Boolean Algebra unary operators tend to have higher priority than binary operators. This will help you to evaluate $2 - -2$ and then solve the exercise. We postpone the problem how to identify unary and binary minus sign to a series of exercises in subsection 9.2.4.

For the time being we do not bother how constants or variables are generated and temporarily take V as a terminal symbol. A grammar G taking care of the above problems has the following terminal symbols

$$G.T = \{+, -_2, *, /, -_1, (,), V\}$$

For each of the priority levels we have a nonterminal symbol

$$G.N = \{F, T, A\}$$

F like 'factor' stands for the highest priority level of the unary minus or an expression in brackets

$$F \to V| -_1 F|(A)$$

T like 'term' stands for the priority level of multiplication and division

$$T \to F|T * F|T/F$$

A like 'arithmetic expression' is the start symbol and stands for the priority level of addition and subtraction

$$A \to T|A + T|A -_2 T$$

We illustrate this grammar by a few examples: in figure 9.12 we have a derivation tree for $V/V/V/V$. This tree suggests evaluation of the expression from left to right. For our first exercise above we would evaluate

$$2/2/2/2 = ((2/2)/2)/2 = 1/4$$

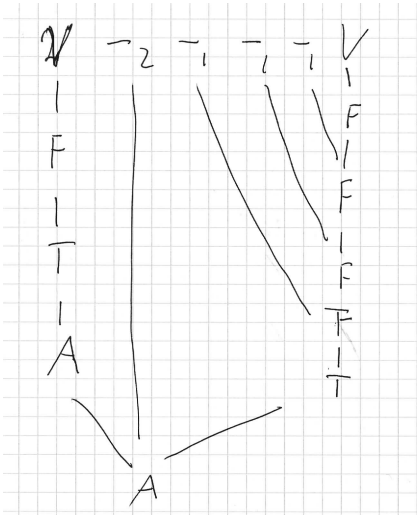


Figure 9.13: A derivation tree reflecting the usual priority rules

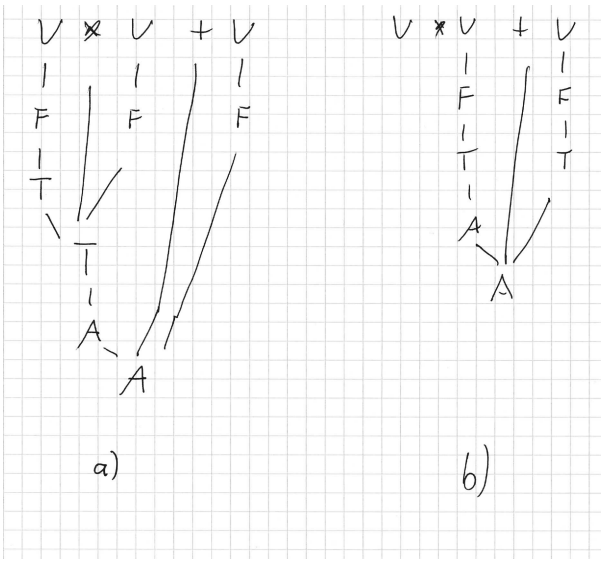


Figure 9.14: A derivation tree reflecting the usual priority rules

In figure 9.13 we have a derivation tree for $V -_2 -_1 -_1 -_1 V$. According to this tree we would evaluate

$$2 - - - -2 = 2 - (-(-(-2))) = 4$$

We postpone the problem how to distinguish between unary and binary minus until later. In figure 9.14 a) we have a tree for $V * V + V$ suggesting that * binds more strongly than +. An attempt to start the construction of a derivation tree as in figure 9.14 b) apparently leads into a dead end. Indeed one can show for the above grammar G

Lemma 67. *G is unambiguous.*

This lemma is nothing less than fundamental, because it explains, why expression evaluation is well defined. The interested reader can find the proof in the next subsection.

9.2.3 Proof of Lemma 67

We proceed in 4 fairly obvious steps, showing successively that derivation trees are unique for

1. factors without brackets

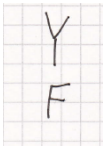


Figure 9.15: Deriving V from F

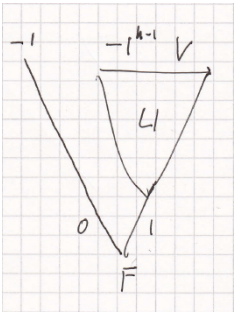


Figure 9.16: Decomposition of a derivation tree T_n for a border word with $n \geq 1$ unary minus signs from F

2. terms without brackets

3. arithmetic expressions without brackets

4. arithmetic expression with brackets

Lemma 68. *Let $F \to^* w$ and $w \in \{V, -_1\}^+$. Then the derivation tree T of w from F is unique*

Proof. Fist observe that production $F \to (A)$ cannot be used, because w contains no brackets, productions $T \to T \circ F$ and $A \to A \circ T$ cannot be used, because $*, /, +, -_2$ do not occur, and we are left with productions

$$F \to V | -_1 F \quad T \to F \quad A \to T$$

Now we prove the lemma by induction over the number n occurrences of the unary minus. For $n = 0$ we have $w = V$ and only production $F \to V$ can be used to derive the V as shown in figure 9.15. The symbol F shown in that figure must be at the root of the tree. Otherwise there would be an ancestor of F which is labeled F. The production $F \to -_1 F$ would produce a unary minus, thus it cannot be used. Thus with the productions available ancestors of a node labeled F can only be labeled with T and A.

For the induction step let $n > 0$ let T_n be a derivation tree for $-_1^n V$ and consider the *first* occurrence of symbol $-_1$ in w. It can only be derived by production $F \to -_1 F$ as shown in figure 9.16. We claim that the node labeled with F at the bottom of this figure is the root. If it would have a father labeled F, then the $-_1$ in the figure would not be the leftmost one. Other labels of ancestors can only be T and A.

Decompose the derivation tree T_n into $U = sub(T_n, 1)$ and the rest (that we just have determined) as shown in the figure. U is unique by induction hypothesis. Hence the composition is unique. By lemma 66 this composition is T_n.

□

Lemma 69. *Let $T \to^* w$ and $w \in \{V, -_1, *, /\}^+$. Then the derivation tree T_n of w from T is unique*

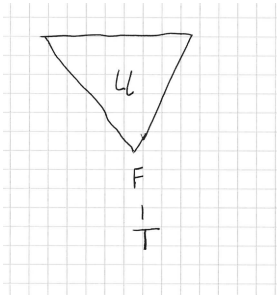


Figure 9.17: Deriving a border word without multiplication and division signs from T

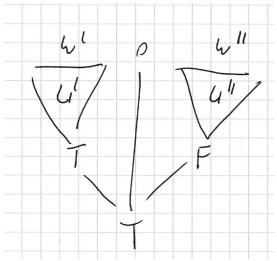


Figure 9.18: Deriving a border word with $n \geq 1$ multiplication and division signs from T

Proof. We can only use productions

$$\begin{aligned} F &\rightarrow V| -_1 F \\ T &\rightarrow F|T * F|T/F \\ A &\rightarrow T \end{aligned}$$

because the other productions produce terminals not in the border word of T_n.

We prove the lemma by induction over the number n of occurrences of symbols $*$ and /. For $n = 0$ the derivation must start at the root labeled T with an application of production $T \rightarrow F$ as shown in figure 9.17 because other productions would produce terminals $*$ or / not in the border word. The remaining part U of the tree is unique by lemma 68

For $n > 0$ consider the *last* occurrence $\circ$ of a multiplication or division sign. Then $w = w' \circ w$". It can only be generated by production $T \rightarrow T \circ F$ as shown in figure 9.18. We claim that the lower node in the figure labeled T is the root. If it has a father labeled T, then $\circ$ in the picture is not the rightmost one in the border word. Thus ancestors can only be labeled with A. Subtree U" is unique by lemma 68. Subtree U" is unique by induction hypothesis. □

Lemma 70. *Let $T \rightarrow^* w$ and $w \in \{V, -_1, *, /, +, -_2\}^+$. Then the derivation tree of w from T is unique*

Proof. We can only use productions

$$\begin{aligned} F &\rightarrow V| -_1 F \\ T &\rightarrow F|T * F|T/F \\ A &\rightarrow T|A + T|A -_2 T \end{aligned}$$

We prove the lemma by induction over the number n of occurrences of symbols $+$ and $-_2$. For $n = 0$ the derivation must start at the root labeled A with application of production $A \rightarrow T$ as shown in figure 9.19 because other productions would produce terminals $+$ or $-_2$ not in the border word. The remaining part U of the tree is unique by lemma 69

For $n > 0$ consider the *last* occurrence $\circ$ of an addition or binary minus sign. Then $w = w' \circ w$". It can only be generated by production $A \rightarrow A \circ T$ as shown in figure 9.20. We claim that the lower

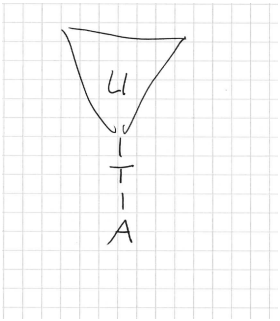


Figure 9.19: Deriving a border word without addition and subtraction signs from A

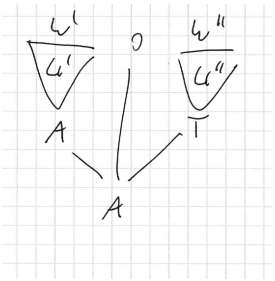


Figure 9.20: Deriving a border word with $n \geq 1$ addition and subtraction signs from A

node in the figure labeled A is the root. If it has a father labeled A, then $\circ$ in the picture is not the rightmost one in the border word. Thus there are no ancestors. Subtree U" is unique by lemma 69. Subtree U" is unique by induction hypothesis. □

Now we prove lemma 67 by induction on the number n of bracket pairs. For $n = 0$ we get the uniqueness of the derivation tree from lemma 70. For $n > 0$ decompose

$$w = w^1(w^2)w^3$$

where the pair of brackets shown is an innermost bracket pair, i.e. w^2 contains no brackets. The bracket pair shown is generated by production $F \to (A)$ as shown in figure 9.21. Subtree U' of the derivation tree is unique by lemma 70. Assume there are two different derivation trees U^1 and U^2 deriving w^1Fw^3 from A. Applying production $F \to V$ in both trees to the leaf F shown gives two derivation trees for w^1Vw^3 which has $n - 1$ pairs of brackets. This is impossible by induction hypothesis.

9.2.4 Distinguishing Unary and Binary Minus

The theory of this chapter cannot yet be applied directly to ordinary arithmetic expressions, simply because in ordinary expressions one does not write $-_1$ or -2. One simply writes $-$. Fortunately there is a very simple recipe how to identify a binary Minus: it only stands to the right of symbols V or), whereas the unary Minus never stands to the right of these symbols. The reader can justify this rule in the following series o

exercises

1. if $a \in L(F) \cup L(T) \cup L(A)$ is any expression, then a ends with symbol) or V
2. if $u -_2 v \in L(A)$ is an arithmetic expression, then u ends with) or V
3. if $u -_1 v \in L(A)$ is an arithmetic expression, then u ends with $-_1, +, -_2, *$ or /

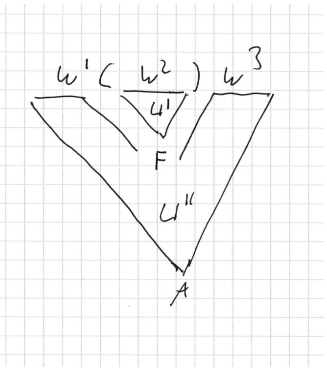


Figure 9.21: Deriving a border word without addition and subtraction signs from A

Chapter 10

The Language $C0$

10.1 Grammar of $C0$

We present syntax and semantics of an imperative programming language called $C0$. In a nutshell it is PASCAL with C syntax. Although $C0$ is powerful enough to do some serious programming work, its context free grammar G shown in figure 10.1 fits on a single page. Start symbol of the grammar is

$$G.S = \langle prog \rangle$$

For the remainder of this chapter and the next chapter this grammar stays fixed. Thus we drop all subscripts G. The language $L(prog)$ generated by the grammar is a superset of all $C0$ programs. Numerous restrictions on the syntax will be defined later when we define the semantics, i.e. the meaning of $C0$ programs. In the context of program semantics the restrictions will appear most natural: they are just the conditions which happen to make the definition of semantics work. In order to give the reader some overview of the language we proceed first with a brief, preliminary and quite informal discussion of the grammar. The language uses well known key words like *int* or *typedef* etc. We treat these as single symbols of the terminal alphabet. Similarly, certain mathematical operators like $!=$ or && are treated like single symbols of the terminal alphabet.

10.1.1 Names and Constants

Symbol $\langle Di \rangle$ generates decimal digits

$$L(Di) = \{0, \ldots, 9\}$$

and symbol $\langle DiS \rangle$ generates sequences of digits

$$L(DiS) = \{0, \ldots, 9\}^+$$

Symbol $\langle Le \rangle$ generates small and capital letters

$$L(Le) = \{a, \ldots, z, A, \ldots, Z\}$$

Symbols $\langle DiLe \rangle$ generates digits, small letters and capital letters. Symbol $\langle DiLeS \rangle$ generates sequences of such symbols

$$L(DiLeS) = \{0, \ldots, 9, a, \ldots, z, A, \ldots, Z\}^+$$

Names are generated by symbol $\langle Na \rangle$. They are sequences of digits and letters starting with a letter

$$L(Na) = L(Le) \circ \{0, \ldots, 9, a, \ldots, z, A, \ldots, Z\}^*$$

$\langle Di \rangle$	$\longrightarrow$	$0 \mid \ldots \mid 9$	digit
$\langle DiS \rangle$	$\longrightarrow$	$\langle Di \rangle \mid \langle Di \rangle \langle DiS \rangle$	digit sequence
$\langle Le \rangle$	$\longrightarrow$	$a \mid \ldots \mid z \mid A \mid \ldots \mid Z$	letter
$\langle DiLe \rangle$	$\longrightarrow$	$\langle Le \rangle \mid \langle Di \rangle$	alphanumeric symbol
$\langle DiLeS \rangle$	$\longrightarrow$	$\langle DiLe \rangle \mid \langle DiLe \rangle \langle DiLeS \rangle$	sequence of symbols
$\langle Na \rangle$	$\longrightarrow$	$\langle Le \rangle \mid \langle Le \rangle \langle DiLeS \rangle$	name
$\langle C \rangle$	$\longrightarrow$	$\langle DiS \rangle \mid \langle DiS \rangle u \mid null$	*int*-/*uint*-/null-pointer-constant
$\langle CC \rangle$	$\longrightarrow$	'!' $\mid \ldots \mid$ ' $\sim$ '	*char*-constant with ASCII code
$\langle BC \rangle$	$\longrightarrow$	$true \mid false$	*bool*-constant
$\langle id \rangle$	$\longrightarrow$	$\langle Na \rangle \mid \langle id \rangle.\langle Na \rangle \mid \langle id \rangle[\langle E \rangle] \mid \langle id \rangle *$	identifier
$\langle F \rangle$	$\longrightarrow$	$\langle id \rangle \mid -_1 \langle F \rangle \mid (\langle E \rangle) \mid \langle C \rangle \mid \langle id \rangle \&$	factor
$\langle T \rangle$	$\longrightarrow$	$\langle F \rangle \mid \langle T \rangle \cdot \langle F \rangle \mid \langle T \rangle / \langle F \rangle$	term
$\langle E \rangle$	$\longrightarrow$	$\langle T \rangle \mid \langle E \rangle + \langle T \rangle \mid \langle E \rangle -_2 \langle T \rangle$	expression
$\langle Atom \rangle$	$\longrightarrow$	$\langle E \rangle > \langle E \rangle \mid \langle E \rangle \geq \langle E \rangle \mid \langle E \rangle < \langle E \rangle \mid$	atom
		$\langle E \rangle \leq \langle E \rangle \mid \langle E \rangle == \langle E \rangle \mid \langle E \rangle \neq \langle E \rangle \mid \langle BC \rangle$	
$\langle BF \rangle$	$\longrightarrow$	$\langle id \rangle \mid \langle Atom \rangle \mid\sim \langle BF \rangle \mid (\langle BE \rangle)$	boolean factor
$\langle BT \rangle$	$\longrightarrow$	$\langle BF \rangle \mid \langle BT \rangle \wedge \langle BF \rangle$	boolean term
$\langle BE \rangle$	$\longrightarrow$	$\langle BT \rangle \mid \langle BE \rangle \vee \langle BT \rangle$	boolean expression
$\langle St \rangle$	$\longrightarrow$	$\langle id \rangle = \langle E \rangle \mid \langle id \rangle = \langle BE \rangle \mid \langle id \rangle = \langle CC \rangle \mid$	assignment statement
		$if\ \langle BE \rangle\ \{\langle StS \rangle\}\ else\ \{\langle StS \rangle\} \mid$	cond. statement
		$if\ \langle BE \rangle\ \{\langle StS \rangle\} \mid$	
		$while\ \langle BE \rangle\ \{\langle StS \rangle\} \mid$	loop
		$\langle id \rangle = \langle Na \rangle (\langle PaS \rangle) \mid$	function call
		$\langle id \rangle = new\ \langle Na \rangle *$	allocation of memory
$\langle rSt \rangle$	$\longrightarrow$	$return\ \langle E \rangle \mid return\ \langle BE \rangle \mid return\ \langle CC \rangle$	return-statement
$\langle PaS \rangle$	$\longrightarrow$	$\varepsilon \mid \langle ParS \rangle$	parameter sequence
$\langle ParS \rangle$	$\longrightarrow$	$\langle E \rangle \mid \langle E \rangle, \langle ParS \rangle \mid \langle BE \rangle \mid \langle BE \rangle, \langle ParS \rangle$	non empty sequence
$\langle StS \rangle$	$\longrightarrow$	$\langle St \rangle \mid \langle St \rangle ; \langle StS \rangle$	statement sequence
$\langle program \rangle$	$\longrightarrow$	$\langle TyDS \rangle ; \langle VaDS \rangle ; \langle FuDS \rangle \mid$	$C0$-program
		$\langle VaDS \rangle ; \langle FuDS \rangle \mid$	no type declaration
		$\langle TyDS \rangle ; \langle FuDS \rangle \mid$	no variable declaration
		$\langle FuDS \rangle$	only function declaration
$\langle TyDS \rangle$	$\longrightarrow$	$\langle TyD \rangle \mid \langle TyD \rangle ; \langle TyDS \rangle$	type declaration sequence
$\langle TyD \rangle$	$\longrightarrow$	$typedef\ \langle TE \rangle\ \langle Na \rangle$	type declaration
$\langle Ty \rangle$	$\longrightarrow$	$int \mid bool \mid char \mid unint \mid \langle Na \rangle$	type names
$\langle TE \rangle$	$\longrightarrow$	$\langle Ty \rangle [\langle DiS \rangle]$	type expression, array
		$\langle Ty \rangle * \mid struct \{\langle VaDS \rangle\}$	pointer, struct
$\langle VaDS \rangle$	$\longrightarrow$	$\langle VaD \rangle \mid \langle VaD \rangle ; \langle VaDS \rangle$	variable declaration sequence
$\langle VaD \rangle$	$\longrightarrow$	$\langle Ty \rangle\ \langle Na \rangle$	variable declaration
$\langle FuDS \rangle$	$\longrightarrow$	$\langle FuD \rangle \mid \langle FuD \rangle ; \langle FuDS \rangle$	function declaration sequence
$\langle FuD \rangle$	$\longrightarrow$	$\langle Ty \rangle\ \langle Na \rangle (\langle PaDS \rangle) \{\langle VaDS \rangle ; \langle body \rangle\} \mid$	function declaration
		$\langle Ty \rangle\ \langle Na \rangle (\langle PaDS \rangle) \{\langle body \rangle\}$	no local variables
$\langle PaDS \rangle$	$\longrightarrow$	$\varepsilon \mid \langle ParDS \rangle$	parameter declaration sequence
$\langle ParDS \rangle$	$\longrightarrow$	$\langle VaD \rangle \mid \langle VaD \rangle, \langle ParDS \rangle$	non empty
$\langle body \rangle$	$\longrightarrow$	$\langle rSt \rangle \mid \langle StS \rangle ; \langle rSt \rangle$	function body

Table 10.1: Grammar of $C0$

Constants are generated by symbol $\langle C \rangle$. They are decimal representations of numbers, possibly followed by the letter u.

$$L(C) = L(Di)^+ \cup L(Di)^+ \circ \{u\}$$

Without the letter u they specify so called integer constants representable by twos complement numbers of appropriate length. With the letter u they specify so called unsigned integer constants representable by binary numbers of appropriate length. Constants of type character are generated by symbol $\langle CC \rangle$. They are digits or letters enclosed by quotation marks. The two boolean constants generated by symbol $\langle BC \rangle$ are $true$ and $false$. Names are required to be different from key words.

Printable symbols with an ASCII-code are

```
!"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
`abcdefghijklmnopqrstuvwxyz{|}~
```

Using three dots we sometimes write the set of printable ASCII symbols as

$$\{! \ldots \sim\}$$

We have used this in the grammar for the definition of character symbols. We denote the set of *all* ASCII symbols as ASC and thus have

$$L(CC) \subset \{'\alpha' : \alpha \in ASC\}$$

With

$$ascii : ASC \to \mathbb{B}^8$$

we denote the function which maps every ASCII symbol $\alpha \in ASC$ to its ASCII code $ascii(\alpha)$. For later reference we remark that the non printable symbol NUL has ASCII code 0^8

$$ascii(NUL) = 0^8$$

10.1.2 Identifiers

Variables and subvariables are specified by identifiers generated by symbol $\langle id \rangle$[1]. Identifiers are composed of the following five components:

- plain variable names x,
- struct component accesses of the form $x.n$ where x is an identifier and n is the name of a struct component,
- array accesses of the form $x[e]$ where x is an identifier and e is an arithmetic expression,
- memory access $x*$ performed by dereferencing an identifier x, and
- taking address of $x\&$, which results in a pointer to the variable or subvariable specified by identifier x.

The names $.n$ of struct component accesses and the indices $[n]$ of array accesses select subvariables of variables or, if applied repeatedly, of subvariables. We stress the fact that we have *not* defined variables or subvariables yet. Thus far, we have only specified names and identifiers for variables and subvariables.

[1]Except for the address-of operation which is generated by rule $\langle F \rangle \to \langle id \rangle\&$. By this, we ensure that there is only a single address-of operation in any given identifier.

10.1.3 Arithmetic and Boolean Expressions

The productions for the arithmetic expressions were basically discussed in the previous chapter. Factors can now be identifiers and constants.

The productions for boolean expressions have exactly the same structure. Boolean factors are identifiers, boolean expressions in brackets and atoms; atoms are either boolean constants or they have the form $a \circ b$ where a and b are arithmetic expressions and $\circ$ is a comparison operator.

10.1.4 Statements

There are six kinds of statements

- *assignment statements.* They have the form

  $$id = e$$

  where identifier id on the left hand side specifies a variable or subvariable to be updated. On the right hand side e can be an arithmetic expression, a boolean expression or a character constant. The value of the variable or subvariable specified by the current value of id is updated by the current value of e. The reader should recall that at this point of the discussion we have not defined yet what variables or subvariables are. Thus we clearly don't have a formal concept of their current values yet.

- *conditional statements.* They can have the form

  $$if \quad e \quad \{ifpart\} \quad else \quad \{elsepart\}$$

  or

  $$if \quad e \quad \{ifpart\}$$

  where e is a boolean expression and $ifpart$ as well as $elsepart$ are sequences of statements without return statements. If expression e evaluates to $true$ the $ifpart$ is executed. If e evaluates to $fales$ then the $elsepart$ is executed if it exists.

- *while statements.* They have the form

  $$while \quad e \quad \{loop \quad body\}$$

  where e is a boolean expression and $\{loop \quad body\}$ is a sequence of statements without return statements. Expression e is evaluated. If it evaluates to $false$ the loop body is skipped and execution of the while loop is complete. If it evaluates to true, the loop body is executed and the while loop is executed again. Thus the body is repeatedly executed while expression e evaluates to $true$ (which might be forever).

- *function calls.* They have the form

  $$id = f(e_1, \ldots, e_p)$$

  where f is the name of a declared function. Function f is called with parameters given by the current values of expressions e_i. The sequence of parameters can be empty. The value returned by the (return statement of the) function is assigned to the variable or subvariable specified by the current value of identifier id on the left hand side.

- *return statements.* They have the form

$$return \quad e$$

  where e is an arithmetic or boolean expression. Always executed at the end of the execution of a function it returns the current value of expression e.

- *new statements.* They allocate new variables in a memory region called the *heap* and have the form

$$id = new \quad t*$$

  where t is the name of an elementary or declared type. The newly generated variable has type t. A pointer to that variable of type $t*$ is assigned to the variable or subvariable specified by the current value of identifier id.

10.1.5 Programs

Programs have three parts:

- a sequence of type declarations $\in L(TyDS)$,
- a sequence of declarations of so called global variables $\in L(VaDS)$, and
- a sequence of function declarations $\in L(FuDS)$.

Type declarations and declarations of global variables can be absent.

10.1.6 Type and Variable Declarations

A type $x \in L(Ty)$ belongs either to set ET of elementary types

$$ET = \{int, uint, bool, char\}$$

or it is a (declared) type name $x \in L(Na)$.

Types are declared by means of type declaration sequences $\in L(TyDS)$ (which are sequences of type declarations $\in L(TyD)$). A type declaration has the form

$$typedef \quad te \quad x$$

where $x \in L(Na)$ is the declared type and $te \in L(TE)$ is a *type expression* specifying the declared type. Type expressions come in three flavors:

- array type expressions of the form

$$t[z]$$

  specify array types. An array variable X of this type consists of array elements $X[i]$ of type t and its length is specified by (decimal number) z, i.e. we have $i \in [0 : \langle z \rangle_Z - 1]$

- pointer type expression of the form

$$t*$$

  specify pointer types. Variables of type $t*$ are pointers to variables or subvariables of type t.

- struct types of the form

$$struct\{t_1 \quad n_1; \ldots; t_s \quad n_s\}$$

  specify struct types. A struct variable X of this type consists of components $X.n$ where $n \in \{n_1, \ldots, n_s\}$. The type of component $X.n_i$ is type t_i.

Note that, in a well-formed C0 program, type declarations are *non-circular*, i.e. any type declaration uses only type names declared earlier in the type declaration sequence (in addition to the elementary types that may be used freely).

Variable declaration sequences $\in L(VaDS)$ are sequences of variable declarations $\in L(VaD)$ separated by semicolons. A variable declaration has the form

$$t \quad x$$

where t is the type of the declared variable and x is its name. We will see that variable declaration sequences occur in three places: i) in the declarations of the global variables of a program, ii) in the declaration of local variables of a function, and iii) in the declarations of struct components. Later we will be able to treat the first two cases as a special case of the third case. Note that, in a well-formed C0 program, every non-elementary type name used in a variable declaration is declared by a corresponding type declaration.

10.1.7 Function Declarations

Function declaration sequences $\in L(FuDS)$ are sequences of function declarations $\in L(FuD)$ separated by semicolons. A function declaration has the form

$$t \quad f \quad (t_1x_1, \ldots, t_px_p)\{t_{p+1}x_{p+1}; \ldots; t_sx_s; body\}$$

where

- f is the name of the declared function
- t is the type of the result returned by the declared function
- $t_1x_1, \ldots, t_px_p$ is the sequence of parameter declarations. It differs from variable declaration sequences in the profound aspect that parameter declarations are separated by commas instead of colons.
- $t_{p+1}x_{p+1}; \ldots; t_sx_s$ is the sequence of declarations of local parameters
- $body \in L(body)$ is a sequence of ordinary statements $\in L(St)$ followed by a return statement.

10.1.8 Representing and Processing Derivation Trees in $C0$

Later in this text we will present numerous algorithms that work on derivation trees T for the above $C0$ grammar G. In case one wishes to implement these algorithms in $C0$, one has to assume that the tree T is represented as a $C0$ data structure. We describe such a representation. To do that, we prefer to switch temporarily to the clean representation of derivation trees from section **??**. There, we defined derivation trees as a pair $T.A$ and $T.\ell$, where

- $T.A$ is the set of nodes of the tree
- mapping $T.\ell : T.A \to N \cup T$ assigns to each node $i \in T.A$ its label $T.\ell(i)$. In drawings we represent a node i with its label $n = T.\ell(i)$ as $i : n$
- nodes $i \in T.A$ are sequences of integers $(i_1, \ldots, i_s) \in \mathbb{N}^*$
- edges are implicitly coded in the names of the nodes. If node i has k sons, they are included into $T.A$ as $i \circ 0, i \circ (k-1)$, where the last indices specify the order of the sons. Thus if $i \circ x$ is one of these nodes, its brother to the right is $i \circ (x+1)$

Nodes i of a derivation tree T will be stored as struct variables $X(i)$ of type DTE (for derivation tree element). Type DTE and the type of pointers to DTE are declared in the following way

```
typedef DTE* DTEp;
typedef struct {uint label; DTEp father; DTEp fson; DTEp bro} DTE;
```

Figure 10.1 illustrates how a node i with nonterminal n and its sons with Nonterminals or Terminals $w[0:k-1]$ are represented by structs of type DTE. We switch to a precise

The interpretation of the components of a struct variable $X(i)$ of type DTE representing node $i = (i_1, \ldots, i_s)$ in the tree is as follows

- $X(i).label$ contains label $T.\ell(i)$ coded as a string in $\mathbb{B}^{32}$
- $X(i).father$ points to the struct variable $X(i_1 \ldots i_{s-1})$ representing the father of node i in $T.A$. If i is the root we make $X(i).father$ the null pointer
- $X(i).fson$ points to the struct variable representing $X(i \circ 0)$, i.e. to the *first* son of i. If i is a leaf we make $x.fson$ the null pointer
- $X(i).bro$ points to the next brother $X(i_1 \ldots i_{s-1} i_s + 1)$ to the right of i in the tree if there is such a brother. Otherwise we make $X(i).bro$ the null pointer

As a first programming example we specify a function $nson$ with two parameters:

- a pointer p to a variable $X(i)$ of type DTE; we assume that X represents a node $i \in T : A$ of a derivation tree T with at least one son.
- a number $j \in \mathbb{N}$ representing a possibly existing son $i \circ j$ of i in T

The function is supposed to return a pointer to $X(i \circ j)$, i.e. to the struct variable representing the son number j of i, if such a son $i \circ j$ exists and the null pointer otherwise. In the implementation the 'first son' component $X.fson$ of X is dereferenced in order to let p point to (a struct representing) the first son of X. In the while loop 'brother pointers' are chased and j is decremented as long as there is a brother and $j > 0$. If the loop ends with $j = 0$ the son $i \circ j$ exists and p points to the struct variable $X(i \circ j)$ representing this son. Otherwise the null pointer is returned.

```
DTEp nson (DTEp p, uint j)
{ DTEp result
  p = p*.fson;
  while p*.bro != null && j != 0
  {
    p = p*.bro;
    j = j-1
  };
  if j != 0 {result = null} else {result = p};
  return result
}
```

As a second programming example uses a data type LEL for linked lists of unsigned integers specified by

```
typedef LEL* LELp;
typedef struct {uint label; LELp next} LEL;
```

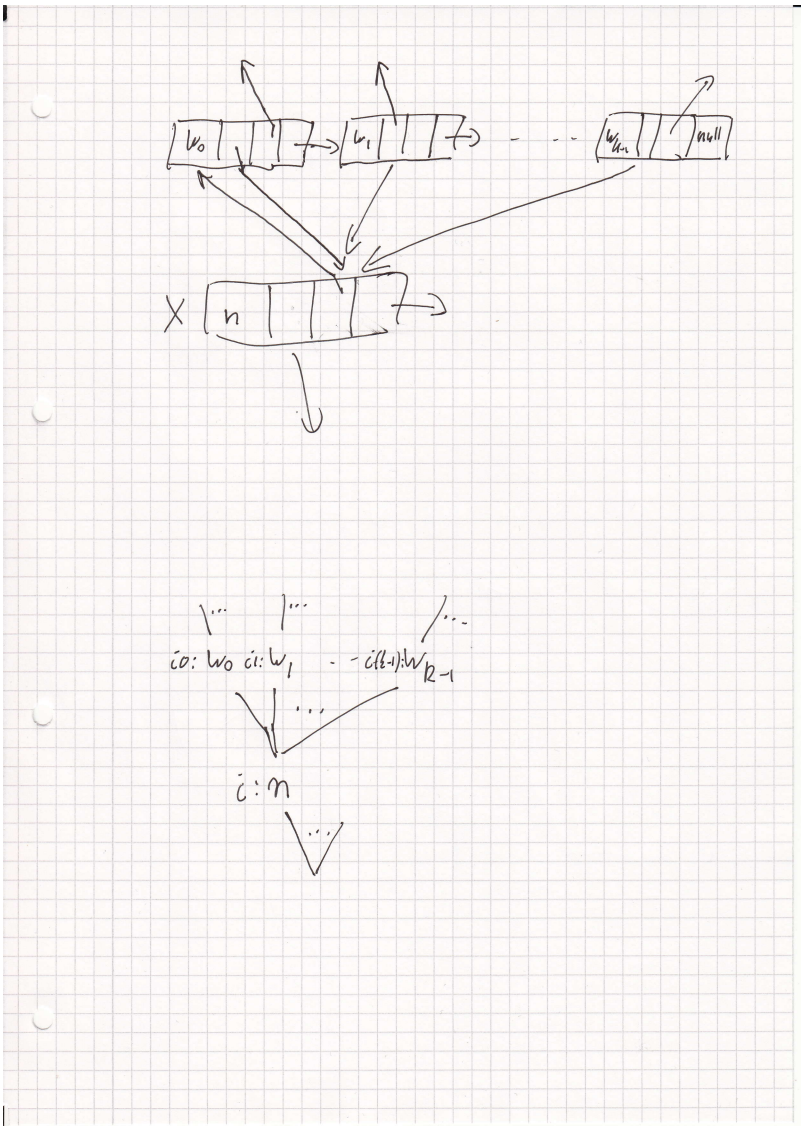


Figure 10.1: Representing a node in a derivation tree and its sons by structs of type DTE

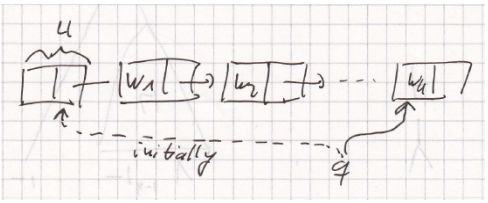


Figure 10.2: Appending a list to a list element U. Pointer q always points to the end of the list.

We specify a function BW which has two parameters:

- a pointer p to a variable $X(i)$ of type DTE representing a node i in the tree.
- a pointer q pointing to a variable U of type LEL

Function BW appends to list element U a linked list, whose *.label* components code the border word $bw(T, i)$ of the subtree with root i. Pointer q always points to the last list element appended so far (see figure 10.2). The implementation quite literally follows the recursive definition of border words for derivation trees from section **??** where we defined

- if i is a leaf of tree T, then its border word is its label:

$$bw(i, T) = T.\ell(i)$$

- the border word of a node i with k sons is obtained by concatenating the border words of its sons

$$bw(i, T) = bw(i \circ 0, T) \circ \ldots \circ bw(i \circ (k-1), T)$$

Thus new list elements are appended whenever the function is called with a parameter p pointing to a leaf X, i.e. when $p* .sons = null$. If X is not a leaf, its sons are processed recursively from left to right until one arrives at a last son Y, which has no brother. The return result is a pointer to the last list element inserted so far.

```
LELp BW (DTEp p, LELp q)
{
  if p*.sons == null
  {
    q*.next = new LEL;
    q = q*.next;
    q*.label = p*.label
  }
  else
  {
    p = p*.fson;
    while p != null
    {
      q = BW(p,q);
      p = p*.bro
    }
  }
  return q
}
```

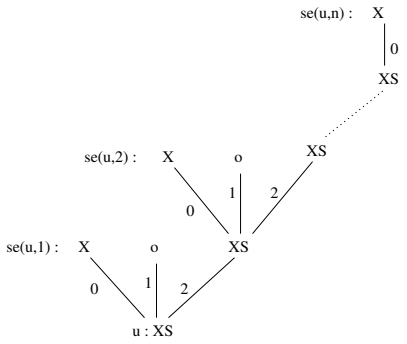


Figure 10.3: Deriving a sequence of elements X at nodes $se[u,i]$ from XS at node u

10.1.9 Sequence Elements and Flattened Sequences in the $C0$ Grammar

The $C0$ grammar has at many places productions for X-sequences of the form

$$\langle XS\rangle \to X \mid X \circ \langle XS\rangle$$

where $\circ$ is a comma or a colon separating the sequence elements X. Figure 10.3 shows a derivation tree with a node u labeled XS and n nodes labeled X. For these nodes we introduce the notation $se(u,i)$ with $\imath \in [1:n]$. We obviously have An easy exercise shows for all i

$$se(u,i) = u2^{i-1}0$$

where 2^i is a sequence of i twos. The *flattened sequence* $u[1:n]$ of the nodes $u[i]$ is denoted by

$$fseq(u) = se(u,[1:n])$$

If we want to advance a pointer p from $X(se(u,i))$ to $X(se(u,i+1))$ we simply follow the father pointer and then select son number 2.

```
p=nson(p.*father,2)
```

10.2 Declarations

We split the presentation of the $C0$ semantics into two parts: static and dynamic semantics. In the section on static semantics we only treat concepts which can be defined by looking at the program without running it. In the section on dynamic semantics we continue the definition of semantics by defining $C0$ configurations c and a transition function δ_C, which computes for $C0$ configuration c the next configuration

$$c' = \delta(c)$$

As usual one then defines $C0$ computations as sequences (c^i) of configurations satisfying

$$c^{i+1} = \delta(c^i)$$

10.2.1 Type Tables

Programs begin with type declaration sequences

$$typedef \quad te_1 \quad n_1; \ldots; typedef \quad te_s \quad n_s$$

where the $t_i \in L(Ty)$ are type expressions and the n_i are the names of the declared types. We collect the set of names of declared types into the set TN

$$TN = \{n_i : i \in [1 : s]\}$$

Type expressions $te \in L(Ty)$ can be of the form $t'[w]$ or $t'*$ where t' is a type name or of the form $struct\{t_1 \quad n_1; \ldots; t_s \quad n_s\}$ where the t_i are type names. We transform such expressions t into a mathematically more convenient form. For declarations of struct types we delete the key word *struct*. For array type declarations we replace decimals number $w[k-1:0]$ by the natural number

$$\langle w \rangle_Z = \sum_{i=0}^{k-1} w_i \cdot Z^i$$

where Z is the number of toes of humans.

$$\tau(te) = \begin{cases} \{t_1 \quad n_1; \ldots; t_s \quad n_s\} & te = struct\{t_1 \quad n_1; \ldots; t_s \quad n_s\} \\ t'[\langle w \rangle_Z] & te = t'[w] \\ te & te = t'* \end{cases}$$

The type table tt simply maps the names n_i to the encoding $\tau(t_i)$ of the defining type expressions.

$$tt(n_i) = \tau(t_i)$$

It is extremely tempting to abbreviate $tt(n_i) = \tau(t_i)$ to $n_i = \tau(te_i)$ Thus the type declaration for list elements and their pointers in an earlier example can be written as

$$\begin{array}{rcl} LELp & = & LEL* \\ LEL & = & \{uint \quad label; LELp \quad next\} \end{array}$$

We will give in to this temptation because it simplifies notation. *However*, the only equations we will ever write are of the form

$$e = e'$$

where the left hand side is an expression evaluating to a type name $t \in ET \cup TN$ and the right hand side is a type table entry. The equations $e = e'$ of this form are simply abbreviations defined by

$$e = e' \leftrightarrow tt(e) = e'$$

We will *not* do or attempt to justify any computations of the form

$$LEL = \{uint \quad label; LEL* \quad next\}$$

or

$$LEL = \{uint \quad label; \{uint \quad label; LEL* \quad next\}* \quad next\}$$

We call a type $t = t'*$ a pointer type; we call a type t *simple* if it is elementary or a pointer type.

$$\begin{array}{rcl} pointer(t) & \leftrightarrow & \exists t' : t = t'* \\ simple(t) & \leftrightarrow & t \in ET \vee pointer(t) \end{array}$$

Otherwise, i.e. if it is an array or struct type, we call it *composite*.

An obvious context condition requires types to be uniquely defined

Context Condition 1.

$$i \neq j \rightarrow n_i \neq n_j$$

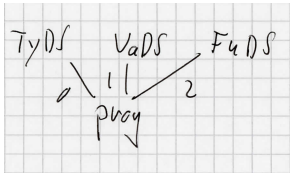


Figure 10.4: Labels $TyDS$, $VaDS$ and $FuDS$ are at nodes 0, 1 and 2 of the derivation tree.

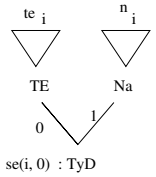


Figure 10.5: Name n_i and type expression te_i of the i'th declared type are the border words of nodes $0[i]1$ and $0[i]1$

A second context condition requires that declarations of composite types only uses elementary or previously defined types.

Context Condition 2. *If t_i is an array type, we require t' to be elementary or to be previously defined*

$$t_i = t'[n] \rightarrow t' \in ET \vee \exists j < i : n_j = t'$$

If t_i is a struct type, we require the types of components to be previously defined

$$t_i = \{t_1 \quad n_1, \ldots, t_s \quad n_s\} \rightarrow \forall k : (t_k \in ET \vee \exists j < i : t_k = n_j)$$

Pointer must point to elementary types or to types declared elsewhere but not necessarily previously

$$n_i = t'* \rightarrow t \in ET \vee \exists j : t' = n_j$$

Indeed in all our examples we declared type DTE after $DTEp$ and LEL after $LELp$. We explicitly encourage readers to feel slightly worried about future consequences of this definition. This is not an ordinary recursive definition where we define new types strictly from known types, and this will indeed produce certain technical complications. Unfortunately, such a definition cannot be avoided, because we wish to have struct types X, which have components $X.n$ pointing to variables of type X.

Implementation hints A programmer wishing to extract the names n_i from the derivation tree should observe that label $TyDS$ is at node 0 of the tree (see figure 10.4). Sequence element j with label TyD is at node $0[j]$; the name n_i can be found as the border word of son 1 of $0[i]$ (see figure 10.5) and the corresponding type expression te_i as the border word of son 0 of $0[i]$. Thus

$$\begin{aligned} n_j &= bw(0[j]1) \\ te_j &= bw(0[j]0) \end{aligned}$$

Computing a list of these names is an easy exercise with the small portions of $C0$ programs given so far. Inspection of the labels of sons 1 of node $1[j]0$ allows to test, what kind of a type is defined: a struct if $\ell(1[j]01) = \{$, an array if $\ell(1[j]01) = [$ and a pointer if $\ell(1[j]01) = *$. If one were to program a data structure for the type table one would store the binary representation of $\langle w \rangle Z$; thus one would have to write a conversion algorithm from decimal to binary numbers.

10.2.2 Global Variables

In programs the type declaration sequence is followed by a variable declaration sequence

$$t_1 \quad x_1; \ldots t_r \quad x_r$$

for the global variables. We collect the names of the global variables in a set

$$VN = \{x_1, \ldots, x_r\}$$

Except for the missing keyword *struct* and the brackets this looks very much like the declaration of a struct type. Indeed it turns out that both in semantics and in compiler construction global variables x are treated *exactly* like components $gm.x$ of a struct variable gm for the global memory. We therefore introduce for the type of this variable gm a new name $\$gm$, extend the range of the type table to include gm and define

$$tt(\$gm) = \{t_1 \quad x_1; \ldots t_r \quad x_r\}$$

resp. shorter

$$\$gm = \{t_1 \quad x_1; \ldots t_r \quad x_r\}$$

10.2.3 Function Tables

The sequence of function declarations of a program has the form

$$t_1 \quad f_1 \quad d_1; \ldots; t_i \quad f_k \quad d_k$$

where for all$i$: t_i is the type of the returned result, f_i is the name of the declared function and d_i is everything else. We collect the names of declared functions into a set

$$FN = \{f_1, \ldots f_k\}$$

and create a function table ft mapping function names $f \in FN$ to *function descriptors* $ft(f)$ which consist of several components $ft(f).y$. In order to specify these components and what is recorded there we expand a function declaration to

$$t \quad f(t_1 \quad x_1, \ldots, t_p \quad x_p) t_{p+1} \quad x_{p+1}; \ldots t_r \quad x_r; body$$

We record in the function table $ft(f)$ the following information

- the type of the result
$$ft(f).t = t$$
- the set of declared parameters and local variables
$$ft(f).VN = \{x_1, \ldots, x_r\}$$
- the number of parameters
$$ft(f).p = p$$
- the body *body* of function f
$$ft(f).body = body$$

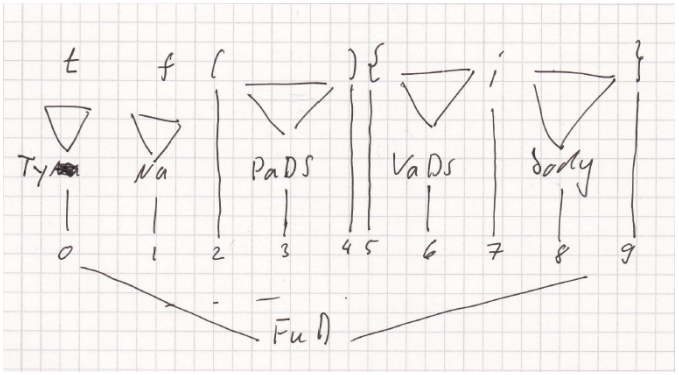


Figure 10.6: The body of a function with a non empty variable declaration sequence is generated by son number 8

Implementation hints: Although quite intuitive, this is not yet a complete definition. It is easily made precise. The function declaration sequence is generated by node 2 (figure x) of the entire derivation tree. If $f = f_i$ then its declaration is generated by node $se(2, i)$, and figure 10.6 shows that the body is generated by son 8 of $se(2, i)$ if the variable declaration sequence for the local parameters is present. Otherwise it is generated by son 6. The parameter declaration sequence is absent if the label of son 6 is *body*. Thus the root $nbody(f_i)$ of the derivation tree of the body of f_i is

$$nbody(f_i) = \begin{cases} se(2,i)8 & \ell(se(2,i)6) \neq body \\ se(2,i)6 & \text{otherwise} \end{cases}$$

and the body of f is the border word of this node

$$body(f_i) = bw(nbody(f_i)$$

Basically we are using the edge labels of the derivation tree as a table of contents of the program text. When people read in a table of contents that the body of function f_i is found in chapter 2, section i, subsection 8 they tend to find nothing wrong with this. Readers who nevertheless find this 'too technical' are strongly encouraged to look for a *precise* definition of the body of the i'th function of a program in less than 3 short lines.

In the same spirit the name f_i of the $i'th$ declared function can be obtained from the derivation tree as

$$f_i = bw(2[i]1)$$

When programs start running, they start by executing the body of a parameterless function *main*. In order for this to work function *main* has obviously to be defined.

Context Condition 3.

$$main \in FN \qquad f_i = main \rightarrow \ell(se[2,i]6) = body$$

Also functions with names f_i should not be defined twice.

Context Condition 4.

$$i \neq j \rightarrow f_i \neq f : j$$

During program execution, every call of function f will create a so called *function frame* for f containing a new copy of the parameters and local variables of the function. We proceed as we did in the case of global memory and treat all parameters and local variables x_i of a function frame for f as components of a single struct variable. Thus we record in the type table

$$\$f = \{t_1 \quad x_1, \ldots, t_p \quad x_p, t_{p+1} \quad x_{p+1}, \ldots, t_r \quad x_r\}$$

that we record in the type table.

10.2.4 Variables and Subvariables of $C0$ Configurations

Recall that - for the purpose of defining semantics - we treat global variables as components of the global memory and we treat parameters and local variables of functions as components of function frames. The only other $C0$ variables are created by the *new* statement; they belong to the so called *heap* and have no name. Therefore the set V variables of all $C0$ programs contains at the top level only three kinds of variables (see figure u).

- the global memory frame gm. Thus
$$gm \in V$$
- the heap variables. They have no name, so we simply number them with elements $j \in \mathbb{N}$. Thus
$$\mathbb{N} \subset V$$
- function frames for declared functions f; they belong to the so called em stack. Because functions can be called recursively, several function frames (f, i) can be present for the same function f. Using indices $i \in \mathbb{N}$ and observing that $f \in L(Na)$ is a name we get
$$L(Na) \times \mathbb{N} \subset V$$

Other variables will not occur. Thus we define the set of all possible $C0$ variables as

$$V = \{gm\} \cup \mathbb{N} \cup Ł(Na) \times \mathbb{N}$$

Next, we define the set SV of all possible subvariables in all $C0$ programs. Variables of struct or array type can have subvariables. Subvariables of structs are selected by *selectors* of the form $.n$ where n is a name Subvariables of arrays are selected by selectors of the form $[n]$ with $n \in \mathbb{N}$. We define the set S of all selectors of all $C0$ programs as

$$S = \{.n : n \in L(Na)\} \cup \{[n] : n \in \mathbb{N}\}$$

Subvariables can be of struct or array type too. Thus selectors can be applied repeatedly (to variables of appropriate type). The set of finite sequences of selectors is S^+ and we get

$$VS^+ \subset SV$$

Allowing empty selector sequences ε we consider each variable X as the subvariable

$$X = X \circ \varepsilon$$

Thus we define

$$SV = VS^*$$

10.2.5 Range of Types and Default Values

For elementary or declared types t we define the *range* of t, i.e. the set of values that variables of type t can assume. As the reader will expect, the definition is recursive. The obvious base case are the elementary types

$$ra(t) = \begin{cases} B_{32} & t = uint \\ T_{32} & t = int \\ ASC & t = char \\ \mathbb{B} & t = bool \end{cases}$$

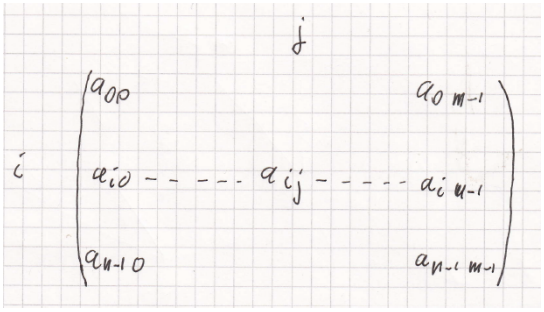


Figure 10.7: Matrices are in $C0$ declared as arrays of rows $a_{i,0}, \ldots, a_{i,m-1}$

It is quite common practice to define in textbooks $ra(int) = \mathbb{Z}$ and $ra(uint) = \mathbb{N}$. This has the advantage of being elegant but also a crucial disadvantage: for real programs running on real computers it is simply wrong. Real computers are finite and can only represent finitely many numbers. In an n bit processor unsigned integer arithmetic is usually modulo 2^n and for n bit unsigned integers we get

$$(2^n - 1) + 1 = 0$$

which violates the Peano axiom, that 0 is not the successor of of any natural number.

For arrays and struct types t we define the set $sel(t)$ of their selectors as

$$sel(t) = \begin{cases} [0 : n-1] & t = t'[n] \\ \{n_1, \ldots, n_s\} & t = \{t_1 \quad n_1; \ldots; t_s \quad n_s\} \end{cases}$$

A value of an array of type $t'[n]$ is then defined as a mapping f assigning to selectors $i \in sel(t)$ values in the range of t'. Values of struct types are mappings from $sel(t)$ to the union of the $ra(t_i)$ with the restriction, that a component $.n_i$ must be mapped into the range of t_i.

$$ra(t) = \begin{cases} \{f | f : sel(t) \to ra(t')\} & t = t'[n] \\ \{f | f : sel(t) \to \cup_i ra(t_i) \wedge \forall i : f(n_i) \in ra(t_i)\} & t = \{t_1 \quad n_1; \ldots; t_s \quad n_s\} \end{cases}$$

Example: we can declare the type of an n by m matrix of integers such that we get from the type table

$$\begin{aligned} row &= int[m] \\ matrix &= row[n] \end{aligned}$$

If $g \in ra(matrix)$ is a value of type $matrix$, then for $i \in [0 : n-1]$

$$g(i) = f_i$$

is a value of type row. In order to represent row i of the matrix in figure 10.7, we set

$$f_i(j) = a_{i,j}$$

for $j \in [0 : m-1]$. Then we get for all i and j

$$g(i)(j) = f(i)(j) = a_{i,j}$$

which fortunately matches intuition. Pointers are the null pointer or they point to subvariables. Thus we define the range of pointer types t as

$$ra(t) = SV \cup \{null\} \quad \text{if} \quad pointer(t)$$

and can show

Lemma 71. *Let $TN = \{t_1, \ldots, t_s\}$ be the set of declared types in the order of their declaration. Then for all i the range $ra(t_i)$ is well defined.*

Proof. For elementary types and for pointer types, the definitions are not recursive, thus there is nothing to show. For array and struct types the lemma follows by induction on i. If $t_i = t'[n]$, then we required t' to be elementary, in which case $ra(t')$ is obviously well defined, or to be previously defined, i.e. $t' = t_j$ with $j < n$, in which case $ra(t_j)$ is well defined by induction hypothesis. Similarly, if $t_i = \{r_u \quad n_1; \ldots; r_v \quad n_v\}$ then we required component type r_w to elementary, in which case $ra(r_w)$ is obviously well defined, or to be previously defined, i.e. $r_w = t_k$ with $k < i$, in which case $ra(r_w)$ is well defined by induction hypothesis. □

For every simple type t we define in an obvious way a default value $dft(t) \in ra(t)$. This default value $dft(t)$ is later used to initialize subvariables of type t that are created without an immediate assignment of their value. This will happen to concern all variables except parameters of functions.

$$dft(t) = \begin{cases} 0 & t \in \{int, uint\} \\ NUL & t = char \\ 0 & t = bool \\ null & t = t'* \end{cases}$$

The definition of default values is extended to composite types in an obvious way. The default value of array type $t'[n]$ maps every index i to the default value $dft(t')$ of the array elements.

$$t = t'[n] \wedge i \in [0 : n-1] \rightarrow dft(t)(i) = deft(t')$$

The default value of a struct type $\{t_1 \quad n_1; \ldots; t_s \quad n_s\}$ maps every sector n_i to the default value $dft(t_i)$ of the corresponding component type.

$$t = \{t_1 \quad n_1; \ldots; t_s \quad n_s\} \wedge i \in [1 : s] \rightarrow dft(t)(n_i) = deft(t_i)$$

10.3 $C0$ Configurations

All previous definitions are implicitly part of the definitions made in this section, but as they don't change in the course of computations, we don't list them explicitly in the mathematical formalism. As announced previously we will keep introducing context conditions at the places, where they help in an obvious way to make certain mechanisms work, although they can be checked statically.

10.3.1 Variables, Subvariables and their Type in $C0$ Configurations c

In order to provide intuition for the following formal definitions we sketch the use of the stack and of the concept of recursion depth in the future definitions: computations start with recursion depth 0 and a single function frame $(main, 0)$ on the stack. If a function f is called in a computation of recursion depth rd , then frame $(f, rd + 1)$ is put on the stack and $rd + 1$ is made the new recursion depth. Conversely, if a return statement is executed in a computation of recursion depth rd, then the top frame (with second component rd) is removed from the stack and the recursion depth is decreased by 1. A frame (f, i) on the stack is called a frame at recursion depth i. $C0$ configurations have among others the following components (see figure 10.8)

- the recursion depth

$$c.rd \in \mathbb{N}$$

It counts the number of functions that have been called and that have not returned yet.

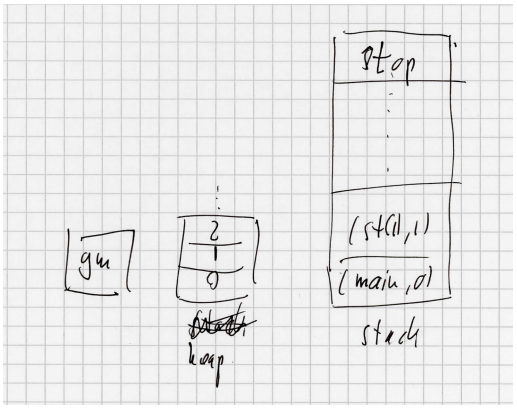


Figure 10.8: Top level variables in the $C0$ semantics: the global memory frame gm, the heap variables numbered $0, 1, \ldots$, and the function frames $(st(i), i)$ of the stack. The bottom frame of the stack belongs to function $main$.

- a stack function
$$c.st : [0 : c.rd] \to FN$$
specifying for each recursion depth $i \in [0 : c.rd]$ the name of the function $c.st(i) \in FN$ such that stack frame
$$ST(c, i) = (c.st(i), i)$$
is the frame with recursion depth i on the stack. We will see later that we always have $c.st(0) = main$, i.e. the bottom frame of the stack is always $(main, 0)$ The top frame, i.e. the frame with maximal recursion depth on the stack is denoted by
$$top(c) = ST(c, c.rd)$$
and the function $c.st(c.rd)$ belonging to the maximal recursion depth on the stack is called the current function $cf(c)$ of configuration c
$$cf(c) = c.st(c.rd)$$

- a result destination function
$$rds : [1 : c.rd] \to SV$$
specifying for every recursion depth $i \geq 1$ the subvariable $c.rds(i)$, where the result of function $c.st(i)$ is returned.

- the number
$$c.nh \in \mathbb{N}$$
of variables on the heap. Recall that we refer to variables on the heap simply by numbers. Thus the set of heap variables in configuration c is $[0 : c.nh - 1]$

- a function
$$c.ht : [0 : c.nh - 1] \to ET \cup TN$$
assigning to each heap variable j its type $c.ht(i)$, which is either elementary or declared.

With these definitions we can define the set $V(c)$ of (top level) variables of a configuration c as the global memory frame, the stack frames and the heap variables in c:
$$V(c) = \{gm\} \cup [0 : c.nh - 1] \cup \{ST(c, i) : i \in [0 : c.rd]\}$$

The *variable type* of $vtype(x, c)$ of variable x in configuration c is defined in the obvious way. The type of the global memory frame gm is $\$gm$ as defined in the type table. Similarly, the type of a function frame (f, i) is $\$f$ as defined in the type table. The type of a heap variable j is defined by the heap type function as $c.ht(j)$

$$vtype(x, c) = \begin{cases} \$gm & x = gm \\ \$f & x = (c.st(i), i) \wedge c.st(i) = f \\ c.ht(x) & x \in [0 : c.nh - 1] \end{cases}$$

We define the set of subvariables $SV(c)$ of configuration c and the *variable type* $vtype(x, c)$ of such subvariables $x \in SV(c)$ by the obvious recursive definitions.

- variables x of configuration c are subvariables of configution c and their variable type $vtype(x, c)$ is already defined
$$V(c) \subset SV(c)$$
- if $x \in SV(c)$ is a subvariable of array type, then elements $x[i]$ are subvariables with the corresponding element type
$$vtype(x, c) = t'[n] \rightarrow \forall i \in [0 : n - 1] : x[i] \in SV(c) \wedge vtype(x[i], c) = t'$$
- if $x \in SV(c)$ is a subvariable of struct type, then components $x.n_i$ are subvariables with the corresponding element type
$$vtype(x, c) = \{t_1 \quad n_1; \ldots t_s \quad n_s\} \rightarrow \forall i \in [1 : s] : x.n_i \in SV(c) \wedge vtype(x.n_i, c) = t_i$$

A subvariable $x \in SV(c)$ is called *simple* if its variable type is simple; it is called a pointer if its variable type is a pointer type.

$$\begin{aligned} simple(x, c) &\equiv simple(vtype(x, c)) \\ pointer(x, c) &\equiv pointer(vtype(x, c)) \end{aligned}$$

Subvariables $x \in SV(c)$ have the form $x = ys$ where $x \in V(c)$ is a variable and $s \in S^*$ is a selectors sequence. Subvariables ys can belong to the global memory, the heap or the stack iff the corresponding variable y belongs to the global memory, heap or stuck. Thus we define for $x \in SV(c)$predicates

$$\begin{aligned} ingm(x, c) &\equiv \exists s \in S^* : x = gm.s \\ onheap(x, c) &\equiv \exists i \in \mathbb{N}, s \in S^* : x = is \\ onstack(x, c) &\equiv \exists i \in \mathbb{N}, s \in S^* : x = (c.st(i), i)s \end{aligned}$$

10.3.2 Value of Variables, Type Correctness and Invariants

Variables and subvariables have types

$$t \in ET \cup TN$$

Subvariables of type t should assume values in $ra(t)$. Thus the set

$$VA = \bigcup_{t \in ET \cup TN} ra(t)$$

should include all possible values that variables and subvariables can assume.

The value of variables $x \in V(c)$ is specified by a new 'memory content' component of the configuration

- $$c.m : V(c) \to VA$$

  mapping simply variables x of type t to a value $c.m(x)$

.

We call a configuration c type correct for variables and write $tc - V(c)$ if for variables x of the configuration their value $c.m(x)$ lies in the range of $vtype(x, c)$

$$\begin{aligned} &tc - V(c) \equiv \\ &x \in V(c) \wedge vtype(x, c) = t \to c.m(x) \in ra(t)) \end{aligned}$$

Function $c.m$ is extended to subvariables x by the obvious definitions

$$x \in SV(c) \wedge vtype(x, c) = t \to c.m(x) \in ra(t) \tag{10.1}$$

holds. If $x \in SV(c)$ is a subvariable of array type $t = t'[n]$. Then its value $c.m(x, c) \in ra(t)$ is a mapping

$$c.m(x) : [0 : n - 1] \to ra(t')$$

For the subvariables $x[i]$ we define

$$c.m(x[i]) = c.m(x)(i)$$

Similarly, if $x \in SV(c)$ is a subvariable of struct type $t = \{t_1 \quad n_1; \ldots; t_s \quad n_s\}$. Then its value $c.m(x) \in ra(t)$ is a mapping

$$c.m(x) : \{n_1, \ldots, n_s\} \to \cup_i ra(t_i)$$

For the subvariables $x.n_i$ we define

$$c.m(x.n_i) = c.m(x)(n_i)$$

Type correctness can now be extended to subvariables. We define

$$tc - SV(c) \equiv x \in V(c) \wedge vtype(x, c) = t \to c.m(x) \in ra(t))$$

$$\begin{aligned} &tc - V(c) \equiv \\ &(\forall x \in V(c) : vtype(x, c) = t \to c.m(x) \in ra(t)) \end{aligned}$$

and get by a trivial induction that type correctness - if it holds - is extended to subvariables by the above definition.

Lemma 72.

$$tc - V(c) \to tc - SV(c)$$

Conversely, we can also conclude type correctness of all subvariables from the type correctness of all simple subvariables. We define

$$\begin{aligned} &tc - s(c) \equiv \\ &x \in SV(t) vtype(x, i) \in ET \to c.m(x) \in ra(t))) \end{aligned}$$

and conclude

Lemma 73.

$$tc - s(c) \leftrightarrow tc - SV(c)$$

Proof. As type correctness for simple subvariables is a special case of type correctness, the direction from left to right is trivial. The proof in the other direction is intuitively a straight forward induction, but a little bit of technical work has to be invested for defining the induction scheme used. For this purpose the define the *height* $h(t)$ of a type in the following way: the height of simple types is 1

$$simple(t) \rightarrow h(t) = 1$$

The height of an array type is the height of the element type + 1:

$$t = t'[n] \rightarrow h(t) = h(t') + 1$$

The height of a struct type is the *maximum* of its component types +1

$$t = \{t_1 \quad n_1; \ldots; t_s \quad n_s\} \rightarrow h(t) = 1 + \max h(t_i)$$

Now the obvious induction i shows that type correctness holds for subvariables x with $h(vtype(x, c)) = i$. □

For variables of pointer type t we were forced to define the range $ra(t) = V \cup \{null\}$ in a rather broad way in order to make the definition of $ra(t)$ well defined. We now are able to formulate the requirement, that pointers of type $t'*$ which are not null should point to subvariables of type t'. More precisely: subvariables x of variable type $t'*$ should point to (have as value) a subvariable in $SV(c)$ of variable type t'. We call configurations satisfying this requirement type correct for pointers

$$\begin{aligned} &tc - p(c) \equiv \\ &x \in SV(c) \wedge vtype(x, c) = t' * \wedge c.m(x) \neq null \rightarrow \\ &c.m(x) \in SV(c) \wedge vtype(c.m(x), c) = t') \end{aligned}$$

A pointer variable x with a value $c.m(x) \notin SV(c)$, i.e. outside the subvariables of configuration c is called a dangling pointer and violates type correctness for pointers. In order to show the absence of dangling pointers we will later define (quite restrictive) context conditions which permit us to show that pointers only point to the global memory or the heap. We define the corresponding predicate

$$\begin{aligned} &p - targets(c) \equiv x \in SV(c) \wedge pointer(x, c) \wedge va(x, c) \neq null \rightarrow \\ &(ingm(c.m(x), c) \vee onheap(c.m(x), c)) \end{aligned}$$

We define a configuration c to be *type correct* if it is type correct for subvariables and in the strengthened sense for pointers

$$tc(c) \equiv tc - s(c) \wedge tc - p(c)$$

We will define the transition function δ_C add context conditions such that in the end we will be able to show that all $C0$ computations are type correct

Lemma 74. *Let (c^i) be a c0 computation. Then*

$$\forall i : tc(c^i)$$

The proof will obviously proceed by induction on i showing i) that the initial configuration c^0 is type correct and ii) - once transition function δ_C for $C0$ is defined - that c^{i+1} is type correct, if c^i is type correct. Properties which hold for all configurations c of a computation are called *invariants*, and lemma 74 can be reformulated as

Invariant 1. *tc(c)*

We will also show

Invariant 2.

$$p - targets(c)$$

Obviously, invariants are simply parts of induction hypotheses about computations (c^i) where the index i has been dropped.

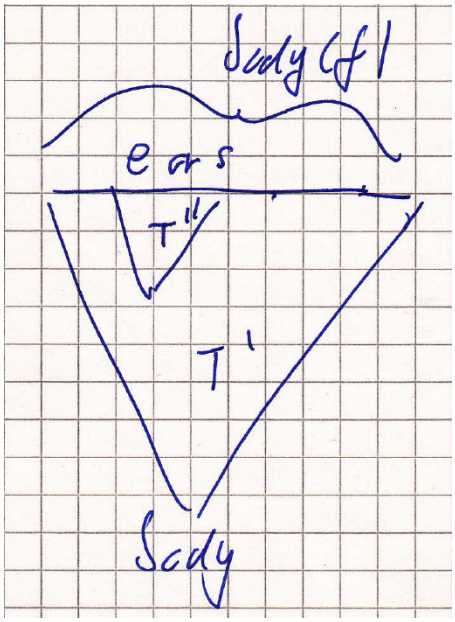


Figure 10.9: Statement s of expression e in the body of function f

10.3.3 Expressions and statements in function bodies

We wish do define when an expression e or a statement s occurs in the body of a function f. Overloading the $\in$-sign we will denote this by

$$e \in f \quad \text{resp.} \quad s \in f$$

Intuitively we consider the subtree T' of the derivation tree whose border word is the body of f, and we say that e or s occurs in f if there is a subtree T" , if e or s is the border word of a subtree T" of T'. This is illustrated in figure 10.9. For the definition of semantics this is precise enough. But later, when we argue about compiler correctness, we need to be able to distinguish between possibly multiple *occurrences* of the same statement or expression (see e.g. figure 10.10). In the full model of derivation trees this is easily done via the roots of the aforementioned subtrees T' We call a node $i \in T.A$ of the derivation tree a *statement node* if it is labeled with St or rSt, and we collect the statement nodes of the tree into the set

$$stn = \{i \in T.A : \ell(i) \in \{St, rSt\}\}$$

Similarly we define the set en of *expression nodes* as

$$en = \{i \in T.A : \ell(i) \in \{CC, BC, id, F, T, E, Atom, BF, BT, BE\}\}$$

We show that every occurrence of an expression is part of an occurrence of a statement, and that every occurrence of a statement is part of a function body, which has been declared in the function declaration sequence. Formally

Lemma 75.

- *Every expression node i has an ancestor (prefix) j which is a statement node*

$$i \in en \rightarrow \exists j \in stn : prefix(j, i)$$

- *Every statement node j has a predecessor k which is labeled with FuD*

$$j \in stn \rightarrow \exists k : prefix(k, j) \wedge \ell(k) = FuD$$

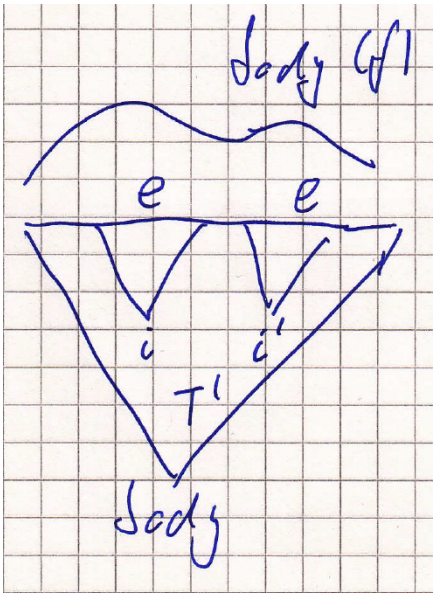


Figure 10.10: Multiple occurrences of the same expression in a function body

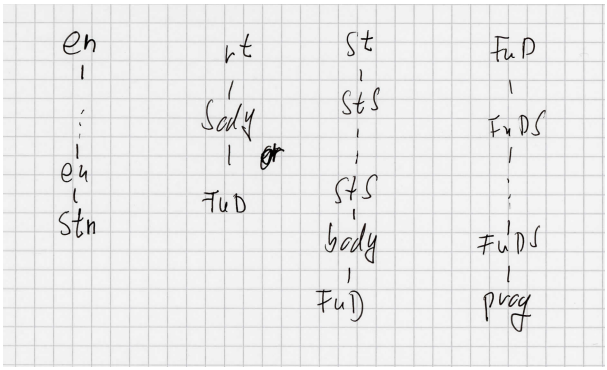


Figure 10.11: Sequences of labels on paths in the derivation tree. Every expression or statement node i has a unique ancestor $Fud(i)$ labeled FuD, which produces the corresponding function declaration.

- *ancestors r of nodes k labeled FuD are not labeled FuD*

$$\ell(k) = body \wedge prefix(r,k) \rightarrow \ell(r) \neq body$$

Proof. Inspection of the grammar shows that fathers of expression nodes are either expression nodes or statement node (see figure 10.11). This shows the first statement. Fathers of statement nodes are labeled StS or *body*; fathers of nodes labeled StS are labeled StS or *body*. Fathers of nodes labeled body are labeled FuD. This shows the second statement. Fathers of nodes labeled FuD are labeled $FuDS$. Fathers of nodes labeled $FuDS$ are labeled $FuDS$ or *prog* and a node labeled *prog* has to father. This shows the third statement. □

Let i be an expression or statement node as shown in figure 10.12. By lemma 75 we can define the unique ancestor $FUD(i)$ of j which is labeled FuD as

$$FuD(i) = \varepsilon\{r : prefix(r,i) \wedge \ell(r) = FuD\}$$

The name $fu(i)$ of the function, to which node i belongs is the border word of son 1 of $FuD(i)$.

$$fu(i) = bw((FuD(i)1))$$

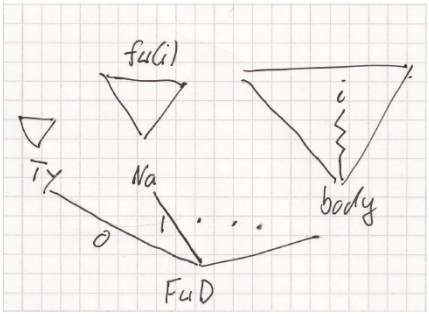


Figure 10.12: Every statement or expression node i belongs to a function body of some declared function $fu(i)$.

For expressions e we define formally, that e occurs in f, if it is the border word of an expression node i with $fu(i) = f$

$$e \in f \equiv \exists i \in en.e = bw(i) \wedge fu(i) = f$$

Similarly, we define for statements s:

$$e \in f \equiv \exists i \in stn.e = bw(i) \wedge fu(i) = f$$

Different occurrences of the same expressionor statement are border words of subtrees with different roots i.

10.3.4 Program Rest

As the next component of $C0$ configurations c we introduce the *program rest* or *continuation* $c.pr$ of configuration c. Program rests are sequences of statements separated by colons, i.e. they have the form

- $$c.pr = c.pr[1:n] = c.pr[1]; \ldots; c.pr[n]) \quad \text{where} \quad \forall i.c.pr[i](i) \in L(St) \cup L(rSt)$$

  The *length* n of the program rest is denoted by $|c.pr|$. The last element of the program rest is denoted by

  $$last(c.pr) = c.pr[|c.pr|]$$

Intuitively speaking this is the sequence of statements and return statements (separated by colons) that is yet to be executed. The statement to be executed in configuration c is the first statement $hd(c.pr)$ of the program rest. Recall that called the top function name $c.st(c.rd)$ on the stack the current function $cf(c)$ of configuartion c:

$$ct(f) = c.st(c.rd)$$

Stack $c.st$ and program rest $c.pr$ are coupled by the crucial

Lemma 76. *The statement $hd(c.pr)$ executed in configuration c belongs to the current function $cf(c)$ of configuration c*

$$hd(c.pr) \in cf(c)$$

Once the definition of the semantics is complete we can show this lemma by induction on the steps i of the computation (c^i). In order to be able to perform the induction step we have to strengthen the induction hypothesis considerably. We first have to show that number of return statements in the program rest equals $c.rd + 1$; intuitively speaking every function frame of the stack belongs to a function whose execution has started and which has not returned yet. Moreover the last statement in a program rest is always a return statement

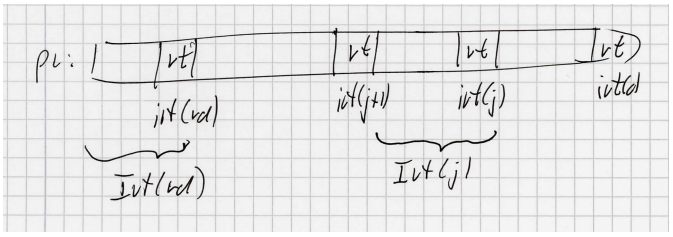


Figure 10.13: Decomposing the indices of the program rest into intervals $Irt(j)$. A statement node $pr(k)$ with $k \in Irt(j)$ belongs to function $st(j)$. Argument c is omitted in this figure.

Invariant 3.

$$\{i : c.pr[i] \in L(rSt)\} = c.rd + 1$$

and

$$last(c.pr) \in L(rSt)$$

We now locate *from right to left* the indices $irt(j, c)$ such that node $c.pr(irt(j))$ is labeled with rSt (see figure 10.13):

$$\begin{aligned} irt(0, c) &= |c.pr|\} \\ irt(j+1, c) &= max\{j : j < irt(j, c) \wedge (c.pr(j)) \in L(rSt)\} \end{aligned}$$

and divide the indices of program rest elements into intervals $Irt(j, t)$ by

$$Irt(j, c) = \begin{cases} [irt(j+1, c) + 1 : irt(j, c)] & j < c.rd \\ [1 : irt(j, c)] & j = c.rd \end{cases}$$

i.e. interval $Irt(j, t)$ extends to the left of return statement j until the next return statement if there is such a statement and otherwise to the head of the program rest. Now we simply formulate an invariant stating that statements $c.pr[k]$ with index k in interval $Irt(, c)$ belong to the function of stack frame $c.st(j)$

$$j \in [0 : c.rd] \wedge k \in Irt(j, c) \rightarrow c.pr[k] \in c.st(j)$$

We collect the above conditions into an invariant about the program rest

Invariant 4. *We say that $inv - pr(c)$ holds if the following conditions are fulfilled*

1.

$$\{i : c.pr[i] \in L(rSt)\} = c.rd + 1$$

2.

$$last(c.pr) \in L(rSt)$$

3.

$$j \in [0 : c.rd] \wedge k \in Irt(j, c) \rightarrow c.pr[k] \in c.st(j)$$

Proving that statement execution maintains this invariant will be very easy once the definition of the semantics is completed.

10.3.5 Result destination Stack

As a last component of $C0$-configurations c we introduce a stack for storing the destination of results to be returned by function calls.

- the return destination stack
$$c.rds : [1 : rd] \to SV(c)$$
stores for all recursion depths $i \in [1 : c.rd]$ the subvariable, where the result of function $c.st(i)$ will be returned.

Clearly, we want the type of the subvariable $c.rds(i)$, where function $f = c.st(i)$ returns its result, to be the type of this result. It is recorded in the function table at $ft(f).t$:

$$vtype(c.rds(i), c) = ft(c.st(i)).t$$

Also it is highly desirable, that the result of the function $c.st(i)$ at recursion depth i is returned to a subvariable, that still exists, after the function has returned. Indeed we will show that entries $x = c.rds(i)$ on the return destination stack are either i) on the heap, or ii) in the gloabal memory or iii) in a subvariable (specified by a selector sequence $s \in S^*$) of function frame $ST(j, c)$ below frame $ST(i, c)$.

$$c.rds(i) = x \to onheap(x, c) \vee ingm(x, c) \vee existss \in S^*, j < i : x = ST(j, c)s$$

We collect these two conditions into an invariant for the return destination stack.

Invariant 5. *We say, that $inv - rds(c)$ holds, if for all $i \in [1 : c.rd]$ the following conditions are fulfilled*

1.
$$vtype(c.rds(i), c) = ft(c.st(i)).t$$

2.
$$c.rds(i) = x \to onheap(x, c) \vee ingm(x, c) \vee existss \in S^*, j < i : x = ST(j, c)s$$

We collect all invariants concerning the well formedness of $c0$-configurations into a single invariant

Invariant 6.

$$inv - conf(c) \equiv tc(c) \wedge ptargets(c) \wedge inv - pr(c) \wedge inv - rds(c)$$

In what follows we aim at a definition of the successor configuration $c' = \delta_C(c)$ of c that permits to show

$$inv - conf(c) \to inv - conf(c')$$

10.4 Initial Configuration

Initial configuration configurations c^0 are defined in the following way: the recursion depth is zero.

$$c^0.rd = 0$$

The only stack frame present is a frame for function *main*.

$$c^0.st(0) = main$$

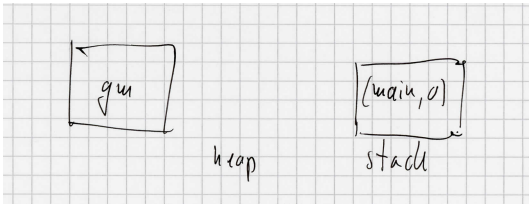


Figure 10.14: Initial configurations $c0$ have only two top level variables: the global memory frame gm and the function frame $ST(0, c) = (main, 0)$. The heap is initially empty.

The heap is empty:

$$c^0.nh = 0$$

and thus there are no arguments for function $c^0.ht$ and we have

$$V(c^0) = \{gm, (main, 0)\}$$

as shown in figure 10.14. Simple subvariables are initialized with their default values. Pointers are null.

$$x \in SV(c^0) \wedge simple(vtype(x, c)) \rightarrow c.m(x) = dft(vtype(x, c))$$

Hence configuration c^0 is type safe for simple subvariables and invariants $tc(c^0)$ and $ptargets(c)$ hold initially. The program rest of c^0 is the body of function $main$

$$c^0.pr = ft(main).body$$

This sequence has a single return statement at the last node, thus the number of returns equals the number of stack frames. Also, all nodes $c^0.pr[k]$ of the program rest belong to function $main$:

$$(\forall k.c^0.pr[]) \in= main) \wedge main = c^0.st(0)$$

Thus invariant $inv - pr(c)$ holds initially. Finally, invariant $inv - rds(c)$ holds trivially, because stack indices $i \geq 1$ are not present. WE summarize

Lemma 77.

$$inv - conf(c^0)$$

10.5 Expression Evaluation

In this chapter we define how to evaluate statements e in configurations c which satisfy invariant $inv - conf(c)$; among other things this will involve determining the type $etype(e, c)$ of expressions e in configurations c. Expression evaluation is part of the semantics of statement execution, and we will later define the first statement $hd(c.pr)$ of the program rest as the statement to be executed in configuration c. By invariant $inv - pr(c)$ this statement belongs to the function $cf(c)$ currently executed in function c:

$$hd(c.pr) \in cf(c)$$

Thus at first glance it suffices to define expression evaluation for expressions $e \in cf(c)$, and if all we want to do is define the semantics of programs, this is indeed good enough. Also evaluation of e will certainly hinge on the derivation tree of e, particularly in order to handle priorities of operators; and this tree is already a subtree of the entire derivation tree T of the program as long as $e \in f$ for any function $f \in FN$. However defining semantics without doing something with it is like building a car and not driving it. With our semantics we want to do two things:

- show that the compiler we construct is correct. Indeed we will use the $C0$ semantics as a guideline for the construction of the compiler.
- prove that programs written in C0 do what they are supposed to do.

For both applications of the semantics it turns out that we better define expression evaluation in a slightly more general way.

- the code generated by a compiler for an expression depends on the type of the expression. But at compile time we do not know the configurations c in which e will be evaluated. All we know is the function f to which e belongs. Thus we should better be able to define the type of an expression e in function f as a function $etype(e, f)$.
- when we want to argue about the correctness of programs, it is often convenient to consider expressions e, which are formed with the variable names x and identifiers of the program but which are *not* part of the program. Fortunately we can refer to the unique derivation trees T' of arbitrary expressions even if they are not subtrees of the derivation tree T of the program.

Let e be an expression with derivation tree T', and let e' be an expression. For later use we define that e' is a *subexpression* of e and write $e' \in e$, if in the derivation tree T' of e there is a node i labeled NA such that e' is the border word of i in T'.

$$e' \in e \equiv \exists i \in T'.N : T'.\ell(i) = NA \wedge bw(i, T') = e$$

The following small example illustrates that this definition of subexpressions respects the priorities of operators . We have

$$1 \in 1 + 2 * 3 \quad \{mbox\}2 * 3 \in 1 + 2 * 3$$

but

$$1 + 2 \notin 1 + 2 * 3$$

because the derivation tree of $1 + 2 * 3$ does not contain a derivation tree for $1 + 2$ as a subtree.

10.5.1 Type, Right Value and Left Value of Expressions

The following definitions are quite obvious: however we will need a somewhat strong context condition to guarantee the absence of dangling pointers. For expressions functions $f \in FN$, $e \in f$ and and $C0$ configuration c satisfying invariants $inv - conf(c)$ we will simultaneously define three values:

- the *expression type* $etype(e, f)$ of the expression e evaluated in function f.
- the *right value* $va(e, c)$ of $bw(i)$ in configuration c. This is a value in the intuitive sense
- the *left value* $lv(e, c)$ in case e is an identifier, i.e. $e \in L(id)$. In this case $lv(e, c) \in SV(c)$ is the subvariable specified by e in configuration c.

The names 'left value' and 'right value' are motivated by the semantics of assignment statements $x = e$ (where $x \in L(id)$ and $e \in L(E) \cup L(BE) \cup L(CC)$) which will be defined in section 10.6. The 'ordinary' value of the *right* hand side is assigned to the subvariable specified by the *left* hand side.

Left values $lv(e, c)$ and right values $va(e, c)$ of expressions $e \in f$ will only be defined for configurations c, where statements from f are executed, i.e. where

$$cf(c) = f$$

in which case the top frame has the format

$$top(c) = (f, c.rd)$$

Whenever the left value $lv(e, c) \in SV(c)$ of an expression e is defined, its right value is *always* determined by the current memory content of subvariable $lv(e, c)$.

- $$va(i,c) = c.m(lv(i,c))$$

As we proceed with the definitions we show (by induction on the derivation tree of e) that the following invariant $inv - expr(e,c)$ concerning the type correctness and pointer targets of expression hold for all expressions e und consideration

Invariant 7. *We say that invariant $inv - expr(e,c)$ holds, if the following conditions are fulfilled:*

1. *Left values $lv(e,c)$ - if they exist - are subvariables of the current configuration*

$$lv(e,c) \in SV(c)$$

2. *The variable type of the left value of an expression is the expression type of the expression*

$$vtype(lv(e,c)) = etype(e,f)$$

3. *The value of an expression is in the range of its expression type*

$$va(e,c) \in ra(etype(e,f))$$

4. *expressions with a pointer type $t'*$ and a non null value point to a subvariable of c which has type t'*

$$etype(e,f) = t' * \wedge va(e,c) \neq null \rightarrow va(e,c) \in SV(c) \wedge vtype(va(e,c)) = t'$$

5. *non null pointers in expression evaluation point to the global memory or the heap*

$$etype(e,f) = t' * \wedge va(e,c) \neq null \rightarrow ingm(va(e,c),c) \vee onheap(va(e,c),c)$$

If an expression e has a left value $lv(e,c)$, then one only has to check parts 1 and 2 of $inv-expr(c)$. Parts 3 to 5 follow:

Lemma 78. *Assume $inv-cons(c)$, let e have a left value $lv(e,c) \in SV(c)$, and assume $vtype(lv(e,c)) = etype(e,f)$. Then*

1. $$va(e,c) \in ra(etype(e,f))$$

2. $$etype(e,f) = t' * \wedge va(e,c) \neq null \rightarrow va(e,c) \in SV(c) \wedge vtype(va(e,c)) = t'$$

3. $$etype(e,f) = t' * \wedge va(e,c) \neq null \rightarrow ingm(va(e,c),c) \vee onheap(va(e,c),c)$$

Proof. Let $x = lv(e,c)$

1. By definition of $va(c,e)$ and invariant $tc(c)$ we have

$$va(e.c) = c.m(x) \in ra(vtype(x)$$

By hypothesis we have

$$vtype(x) = etype(e,f)$$

Hence

$$va(e,c) \in ra(etype(e,f)$$

2. Let $etype(e, f) = t'*$ and assume $va(e, c) \neq null$. We conclude

$$\begin{array}{rcl} t'* & = & etype(e, f) \\ & = & vtype(x) \quad \text{(hypothesis)} \\ va(e, c) & = & c.m(x) \quad \text{(definition of } va(e, c)) \\ & \in & SV(c) \quad \text{(invariant } tc - p(c)) \\ vtype(va(i, c)) & = & vtype(c.m(x)) \\ & = & t' \quad \text{(invariant } tc - p(c)) \end{array}$$

3. Let $etype(i) = t*$ and $va(e, c) = c.m(x) \neq null$. Invariant $p - targets(c)$ gives

$$ingm(c.m(x), c) \vee onheap(c.m(x), c)$$

resp.

$$ingm(va(e, c), c) \vee onheap(va(e, c), c)$$

□

10.5.2 Constants

Evaluation of constants e is independent of the current configuration c and function f. Left values of constants are not defined. There are four kinds of constants e.

- boolean constants $\in true, false$. We define

$$etype(true, f) = etpe(false, f) = bool$$

and

$$\begin{array}{rcl} va(false, c) = 0 \\ va(false, c) & = & 1 \end{array}$$

- Character constants $'a'$ where a is a digit or a letter. We denote by $ascii(a) \in \mathbb{B}^8$ the ASCII code of a and define

$$etype('a', f) = char \quad \text{and} \quad va('a', c) = ascii(a)$$

- integer constants $e =\in [0 : 9]^+$. We set

$$etype(e, z) = int$$

We identify e with a number $e \in \mathbb{N}$ and take its value $tmod2^{32}$

$$va(e, c) = (etmod2^{32})$$

This leaves e unchanged only if we have

$$\langle e \rangle_Z \in [-2^{31} : 2^{31} - 1]$$

- unsigned integer constants $e =\in [0:9]^+ \circ \{u\}$. We set

$$etype(e,z) = uint$$

We identify e with a number $e \in$ and take its value $\mod 2^{32}$

$$va(e,c) = (e \bmod 2^{32})$$

This leaves e unchanged only if we have

$$e \in [0 : 2^{32} - 1]$$

Invariant 7.3 obviously holds for e. Parts 1 of the invariant concerns left values and does not apply. Parts 4 and 5 of the invariant concern pointers and don't apply either.

10.5.3 Variable Binding

We want to evaluate variable names X in functions f resp. configurations c. Variable names $X \in f$ occurring in the body of function f have to be declared either as global variables $\in VN$ or as parameters or local variables $\in ft(f).VN$

Context Condition 5.

$$X \in f \to X \in ft(f).VN \cup VN$$

For variable names $X \in e$ occurring in expressions $e \not\in nf$, that we want to evaluate for function f even if they do not occur in the body of f, the same context condition applies. Let

$$\begin{array}{rcl} \$f & = & \{t_1 \quad n_1, \ldots, t_s \quad n_s\} \\ \$gm & = & \{t'_1 \quad n'_1, \ldots, t'_r \quad n'_r\} \end{array}$$

We take the expression type $etype(X,f)$ of X from the type table for $\$f$ if X is a parameter or local variable of function f. Otherwise we take it from the type table of $\$gm$

$$etype(X,f) = \begin{cases} \in \{t_i : n_i = X\} & X \in ft(f).VN \\ \in \{t'_i : n'_i = X\} & X in VN \setminus ft(f).VN \end{cases}$$

We *bind* variable name $X \in f$ to a subvariable $lv(i,c) \in SV(c)$ of configurations c satisfying $top(c) = (f, c.rd)$. Thus we know

$$(f, c.rd) \in SV(c)$$

This permits us bind variable name X to a local (sub)variable $top(c).X$ of $fu(i)$, if X is the name of a local variable of function $fu(i)$. Otherwise we bind X to global (sub)variable $gm.X$. THis is illustrated in figure 10.15.

$$lv(X,c) = \begin{cases} top(c).X & X \in ft(f).VN \\ gm.X & \text{otherwise} \end{cases} \tag{10.2}$$

Equation 10.2 is called the *visibility rule* of $C0$ (and of C) because it defines how variables are 'visible' from a function body: the local (sub) variable or parameter $top(c).X$ obscures, if it exists, the global variable $gm.X$.

We have $lv(X,c) \in SV(c)$, thus invariant 7.1 holds for X. If $X \in ft(f).VN$ let $X = n_i$; for the variable type of $lv(X,c)$ we get

$$\begin{array}{rcl} vtype(lv(X,c)c) & = & vtype(top(c).X, c) \\ & = & vtype((f, c.rd).n_i, c) \\ & = & t_i \\ & = & etype(X,f) \end{array}$$

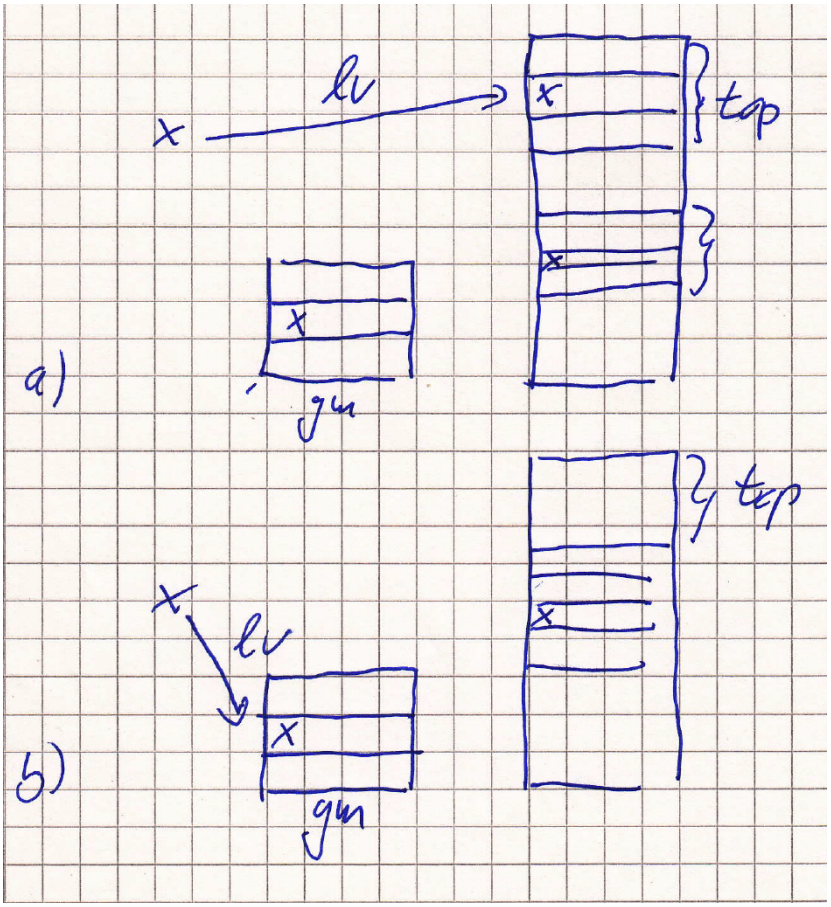


Figure 10.15: Variable name X is bound by the lv function to a local variable or parameter $top(c).X$ in the top frame, if a local variable or parameter with that name exists. Otherwise it is bound to component $gm.X$ with this name in global memory

Otherwise let $X = n'_i$. We get

$$\begin{aligned} vtype(lv(X,c)c) &= vtype(gm.X,c) \\ &= vtype(gm.n'_i,c) \\ &= t'_i \\ &= etype(X,f) \end{aligned}$$

Thus invariant 7.2 holds for X. Parts 3 to 5 follow from lemma 78

10.5.4 Pointer Dereferencing

Let $e = e'*$ where subexpression $e' \in e$ has an expression type $etype(e', f)$. We require that this is a pointer type and we do not dereference null pointers

Context Condition 6.

$$\begin{aligned} etype(e',f) &= t'* \\ va(e',c) &\neq null \end{aligned}$$

Note that the second condition cannot be tested statically. We set

$$\begin{aligned} lv(e,c) &= va(e',c) \\ etype(e) &= t' \end{aligned}$$

Using invariant 7.4 for e' we get

$$\begin{aligned} vtype(lv(e,c)) &= vtype(va(e',c)) \\ &= t' \\ &= etype(e,c) \end{aligned}$$

This is invariant 7.2 for e. Because $va(i0,c) \neq null$ we get from invariant 7.4 for e'

$$lv(e,c) = va(e',c) \in SV(c)$$

This is invariant 7.1 for e. Parts 3 to 5 of the invariant follow from lemma 78

10.5.5 Struct Components

Let $e = e'.n$ where subexpression $e' \in e$ has an expression type $etype(e', f)$. We require that this is a struct type and that n is one of the declared selectors for this type

Context Condition 7.

$$\begin{aligned} etype(i0) &= \{t_1 \quad n_1; \ldots; t_s \quad n_s\} \wedge \\ \exists j : n &= n_j \end{aligned}$$

We define

$$\begin{aligned} etype(e,f) &= t_j \\ lv(e,c) &= (lv(e',c)).n \end{aligned}$$

From invariant 7.1 for e' we know $lv(e',c) \in SV(c)$, hence

$$lv(e,c) = lv(e',c).n \in SV(c)$$

i.e. we have invariant 7.1 for e.

Using invariant 7.2 for e' we get

$$\begin{aligned} vtype(lv(e,c)) &= vtype(lv(e',c).n)) \\ &= t_j \\ &= etype(e,f) \end{aligned}$$

This is invariant 7.2 for e. Parts 3 to 5 of the invariant follow from lemma 78

10.5.6 Array Elements

Let $e = e'[e"]$ where subexpressions $e', e" \in e$ have expression types $etype(e', f)$ resp $etype(e", f)$ We require that $etype(e', f)$ is an array type and that the expression type $etype(e", f)$ of son $e"$ is int or $uint$.

Context Condition 8.

$$\begin{aligned} etype(i0) &= t'[n] \wedge \\ etype(i2) &\in \{int, uint\} \end{aligned}$$

Moreover we require that the current value $va(e", c)$ specifies an index within the array bounds $[0 : n-1]$

Context Condition 9.

$$\langle va(i2,c)\rangle \in [0:n-1] \quad if \quad etype(i2) = uint$$
$$[va(i2,c)] \in [0:n-1] \quad if \quad etype(i2) = int$$

Note that there is no general method to check the latter condition at compile time. We define

$$\begin{aligned} etype(e,f) &= t' \\ lv(e,c) &= lv(e',c)[va(e",c)] \end{aligned}$$

From invariant 7.1 for e' we know $lv(e',c) \in SV(c)$, hence

$$lv(e,c) = lv(e',c).[va(i2,c)] \in SV(c)$$

i.e. we have invariant 7.1 for e.

Using invariant 7.2 for e' we get

$$\begin{aligned} vtype(lv(e,c),c) &= vtype(lv(e',c)[va(e",c)]) \\ &= t' \\ &= etype(e,f) \end{aligned}$$

This is invariant 7.2 for e. Parts 3 to 5 of the invariant follow from lemma 78.

10.5.7 'Address of'

This is slightly more involved. Let $e = e'\&$ be an expression where subexpression $e' \in L(id)$. By the previous subsections identifiers e' have left values in $lv(e',c) \in SV(c)$. Thus e' has the form

$$e' = Xs[1:n]$$

with a variable name

$$X \in VN \cup ft(f).VN$$

For the components $s[i]$ holds

$$s[i] \in \{.n | n \in L(Na)\} \cup \{[a] : a \in L(E)\} \cup \{\}$$

The intention of the following context condition is to forbid, that subvariable $lv(e',c)$ whose address is about to be taken, lies on the stack, because stack frames can disappear from the set of variables on a function return, and this could leave dangling pointers.

Context Condition 10. *We forbid the following situation, which can be checked statically:*

- $$X \in ft(f).VN$$

  i.e. X is the name of a parameter or global variable and hence

  $$lv(X,c) = top(c).X$$

  lies on the stack

- $$\forall i \in [1:n] : s[i] \in \{.n | n \in L(Na)\} \cup \{[a] : a \in L(E)\}$$

  i.e. by the sequence $s[1:n]$ only struct components .n and array elements $[v]$ are computed. Hence $lv(e',c)$ is a subvariable of $top(c).X$ and lies on the stack too.

We define only

$$\begin{aligned} etype(e, f) &= etype(e', f) * \\ va(e, c) &= lv(e', c) \end{aligned}$$

The left value $lv(i, c)$ is not defined. Thus invariants 7.1 and 7.2 do not apply for e.

By invariant 7.1 for e' we have

$$va(e, c) = lv(e', c) \in SV(c)$$

Since

$$SV(c) \subset ra(t*) \quad \text{for any pointer type } t*$$

we have invariant 7.3 for e.

By invariant 7.2 for e' we have

$$vtype(va(e, c)) = vtype(lv(e', c)) = etype(e', c)$$

Thus we have invariant 7.4 for e.

It remains to show invariant 7.5. Reconsider the form of $e' = Xs[1 : n]$. There are three cases:

- X is the name of a local variable and only struct components and array elements are selected. This is the forbidden situation, so we have to prove nothing.
- $X \in VN \setminus ft(f).VN$ is the name of a global variable and only struct components and array elements are selected. We conclude

  $$ingm(lv(e', c), c)$$

  and hence

  $$ingm(va(e, c), c)$$

- there is at least one index j such that $s[i] = *$, i.e. $e" = Xs[1 : j - 1]$ is a pointer that is dereferenced by $s[j]$. Formally

  $$\exists t' : etype(e", f) = t'*$$

  Consider the last (i.e. largest) such index j. By invariant 7.5 for $e"$ we conclude

  $$ingm(va(e", c), c) \vee onheap(va(e", c), c)$$

  By the rules for pointer evaluation we get

  $$va(Xs[1 : j] = va(e"*, c) = lv(e"*, c) = lv(Xs[1 : j]$$

  Hence

  $$ingm(lv(Xs[1 : j], c), c) \vee onheap(lv(Xs[1 : j], c), c)$$

  Because sequence $s[j + 1 : n]$ contains not further components $s[i] = *$ we know, that $lv(e', c)$ is a subvariable of $lv(e", C)$. We conclude

  $$ingm(lv(e', c), c) \vee onstack(lv(e', c), c)$$

  and hence

  $$ingm(va(e, c)c) \vee onstack(lv(e, c), c)$$

  i.e. we have invariant 7.5 for e

10.5.8 Unary Operators

From now on, neither new left values nor new pointers are introduced. Thus only invariant 7.3. is relevant. The trivial proof that it holds is in all remaining cases omitted.

Let $e = \circ e'$ with subexpression $e' \in e$ and unary operator $\circ$.

There are two subcases

- unary minus: $\circ = -_1$. We require

  Context Condition 11.

  $$etype(e', f) \in \{int, uint\}$$

  and set

  $$\begin{aligned} etype(e, f) &= etype(e' f) \\ va(e, c) &= \begin{cases} -va(e, c) tmod 2^{32} & etype(e', f) = int \\ -va(e', c) \bmod 2^{32} & etype(e', f) = uint \end{cases} \end{aligned}$$

  This definition looks 'natural', but if one has no experience with modulo arithmetic one might be up for surprises. Let $etype(e', t) = int$ and

  $$e' = -_1 2147483648 = -2^{31}$$

  We get

  $$\begin{aligned} va(-e', c) &= (2^{31} tmod 2^{32}) \\ &= -2^{31} \\ &= va(e', c) \end{aligned}$$

- negation: $\circ = \neg$. We require

  Context Condition 12.

  $$etype(e', f) = bool$$

  and set

  $$\begin{aligned} etype(i, f) &= bool \\ va(e, c) &= \neg va(e', c) \end{aligned}$$

10.5.9 Binary Operators

Let $e = e' \circ e"$ with subexpressions $e', e" \in e$ and binary operator $\circ$. Recall that subexpressions were defined via subtrees of the derivation tree of e, thus the decomposition of e into e' and $e"$ reflects the priorities of the operators. Because we want to avoid type casting we require here, that subexpressions e' and $e"$ have the same type. For arithmetic operations and comparisons we require this type to be *int* or *uint*, and for boolean operations we require it to be *bool*.

Context Condition 13.

$$\begin{aligned} etype(e', f) &= etype(e", f) \\ (etype(e', f) \in \begin{cases} \{bool\} & \circ \in \{\wedge, \vee, \oplus\} \\ \{int, uint\} & otherwise \end{cases} \end{aligned}$$

The type of the result is the type of the operands for arithmetic operations and *bool* for comparisons or boolean operators:

$$etype(e,f) = \begin{cases} etype(e',f) & \circ \in \{+,-_2,*,/\} \\ bool & \text{otherwise} \end{cases}$$

Now we make the obvious case split.

- addition, subtraction and multiplication: $\circ \in \{+,-_2,*\}$. We define

$$va(e,c) = \begin{cases} (va(e',c) \circ va(e",c) tmod 2^{32} & etype(e',f) = int \\ (va(e',c) \circ va(e",c) mod 2^{32} & etype(e',f) = uint \end{cases}$$

  We have mentioned it before and we stress it again: this is modulo arithmetic, which is very different from arithmetic in $\mathbb{N}$. For instance if

$$\begin{aligned} etype(e,f) &= uint \\ va(e',c) &= 2^{32}-1 \\ va(e",c) &= 1 \\ \circ &= + \end{aligned}$$

  we compute for $va(e,c)$ the binary representation of

$$va(e,c) = (2^{32}-1+1) \mod 2^{32} = 0$$

- division: $\circ = /$. Of course we forbid division by zero

  Context Condition 14.

$$\circ = / \rightarrow va(e",c) \neq 0$$

  For $y \in \mathbb{Z}$ we define $sign(y) \in \mathbb{B}$ as

$$\begin{cases} 0 & y \geq 0 \\ 1 & y < 0 \end{cases}$$

  and have

$$y = (-1)^{sign(y)} \cdot |y|$$

  Let

$$\begin{aligned} y' &= va(e',c) \\ y" &= va(e",c) \end{aligned}$$

  Now we proceed in two steps

  1. we determine the exact result as

$$y = (-1)^{sign(y) \oplus sign(y')} \cdot (|y'|/|y"|)$$

     where / denotes integer division.

2. we determine the result in the usual way

$$va(e,c) = \begin{cases} ytmod2^{32} & etype(e',f) = int \\ ymod2^{32} & etype(e',f) = int \end{cases}$$

Note that integer division *can* overflow (in a nasty way) due to the non symmetric range T_n of two's complement numbers. We already know the relevant example

$$\begin{aligned} e &= -2^{31} \\ e' &= -1 \\ y &= 2^{31} \\ ytmod2^{32} &= -2^{31} \end{aligned}$$

- computation of atoms: $\circ \in \{==, != , <, >, <=, >=\}$ is a comparison operator. We convert the operator in the obvious way into an operator

$$\circ' \in \{=, \neq, >, >, \leq, \geq\}$$

and define

$$va(i,c) = \begin{cases} 1 & va(e',c) \circ' va(e",c) \\ 0 & \text{otherwise} \end{cases}$$

- Boolean operators: $\circ \in \&\&, ||$. We define

$$va(e,c) = \begin{cases} va(e',c) \wedge va(e",c) & \circ = \&\& \\ va(e',c) \vee va(e",c) & \circ = || \end{cases}$$

- Boolean OR.

10.6 Statement Execution

The semantics of statement execution is defined by a case split on the first element $hd(c.pr)$ of the program rest of configuration c. The current configuration is denoted by c; the new configuration is denoted by c'. We assume invariant $inv - conf(c)$ which imply invariant $inv - expr(e,c)$ for the expressions $e \in c.pc$ evaluated in configuration c. Let $f = cf(c)$ be the name of the function executed in configuration c. Definitions will be fairly straight forward. As there are 6 kinds of statements we get the obvious cases:

10.6.1 Assignment

The head of the program rest has the form

$$hd(c.pr) : e = e'$$

We require that the types of left hand hand right hand side of the assignment statement match *and* that these types are simple[2]

Context Condition 15.

$$etype(e,f) = etype(e',f) \wedge simple(etype(e,f)$$

[2]This is a usual restriction in programming languages; relaxing this restriction is a fairly easy exercise.

The execution of the statement does not create or delete variables and it does not manipulate the result destination stack. Formally

$$X \in \{rd, st, rds, nh, ht\} \rightarrow c'.X = c.X$$

which implies

$$SV(c') = SV(c)$$

The memory content $c.x$ of the single simple subvariable $lv(e, c) \in SV(c)$ specified by the left hand side is updated by assigning it the value $va(e', c)$ specified by the right hand side. All other simple subvariables keep their previous value. The assignment statement is dropped from the program rest.

$$\begin{aligned} x \in SV(c) \quad &\wedge \quad simple(vtype(x, c)) \rightarrow \\ (c'.m(x) \quad &= \quad \begin{cases} va(e', c) & x = lv(e, c) \\ c.m(x) & \text{otherwise} \end{cases}) \\ c'.pr \quad &= \quad tail(c.pr) \end{aligned}$$

The set of variables does not change and hence

$$SV(c) = SV(c')$$

A single simple subvariable $lv(e, c)$ is updated. For this subvariable $tc - SV(c')$ follows from invariant 7.3 for the right hand side e' of the assignment. If it is a pointer,$tc - p(c')$ follows from invariant 7.4 for e' and $p - targets(c')$ follows from invariant 7.5. For all other simple subvariables $x \in SV(c) \backslash \{lv(e, c)\}$ the memory content stays the same

$$simple(x, c) \wedge x \in SV(c) \setminus \{lv(e, c)\} \rightarrow c.m(x) = c'.m(x)$$

Thus for these subvariables $tc - SV(c')$, $tc - p(c')$ and $p - targets(c')$ follows by induction from the corresponding invariants for c.

No returns are added to or deleted from the program rest, the recursion depth stays the same, rds is not manipulated. Thus invariant $inv - rds$ is preserved. Dropping the first statement, which is not a return statement from a program rest preserves invariant $inv - pr$.

10.6.2 Conditional Statement

The head or the program rest can have two forms: if-statements

$$hd(c.pr) : ifethen\{if - part\}$$

or if-then-else-statements

$$hd(c.pr) : ifethen\{if - part\}elas\{else - part\}$$

Conditional statements only change the program rest

$$X \neq pr \rightarrow c'.X = c.X$$

Condition e is evaluated. We require it to be of type *bool*.

Context Condition 16.

$$etype(e, f) = bool$$

- if-statements: If the condition evaluates to 1 , the conditional statement is replaced in program rest by the if-part. Otherwise the statement is simply dropped from the program rest.

$$c'.pr = \begin{cases} if - part; tail(c.pr) & va(e,c) = 1 \\ tail(c.pr) & va(e,c) = 0 \end{cases}$$

- if-then-else-statements . If the condition evaluates to 1, the conditional statement is replaced in the program rest by the if-part. Otherwise it is replaced by the else-part.

$$c'.pr = \begin{cases} if - part \circ tail(c.pr) & va(e,c) = 1 \\ else - part \circ tail(c.pr) & va(e,c) = 0 \end{cases}$$

  Thus invariants tc, $p - targets$ and $inv_r ds$ are obviously preserved. Statements s if parts or else parts which are added to the program rest belong to function f, i.e. $s \in cf(c)$. Hence invariant $inv - pr$ is preserved.

subsectionWhile Loop
The head of the program rest has the form

$$hd(c.pr) : whilee\{body\}$$

It changes only the program rest.

$$X \neq pr \rightarrow c'.X = c.X$$

The loop condition e is evaluated. We require it to be of type *bool*

Context Condition 17.

$$etype(e,f) = bool$$

If it evaluates to 0 the while statement is dropped from the program rest. Otherwise the flattened loop body is put in front of the program rest.

$$c'.pr = \begin{cases} tail(c.pr) & va(e,c) = 0 \\ body; c.pr & va(e,c) = 1 \end{cases}$$

There is a certain analogy of this rule with the use of a (potentially limitless) role of toilet paper. One tests if wether one is done and if so one can forget about the entire roll. Otherwise one unrolls a single piece of toilet paper (the loop body) and iterates. Preservation of invariants is shown as for conditional statements

10.6.3 'New' Statement

The head of the program rest has the form

$$hd(c.pr) : newe = t*$$

We require t to be an elementary or declared type. Moreover the subvariable $lv(e,c)$ defined by the left hand side should have type $t*$.

Context Condition 18.

$$t \in TN \cup ET \wedge etype(e,f) = t*$$

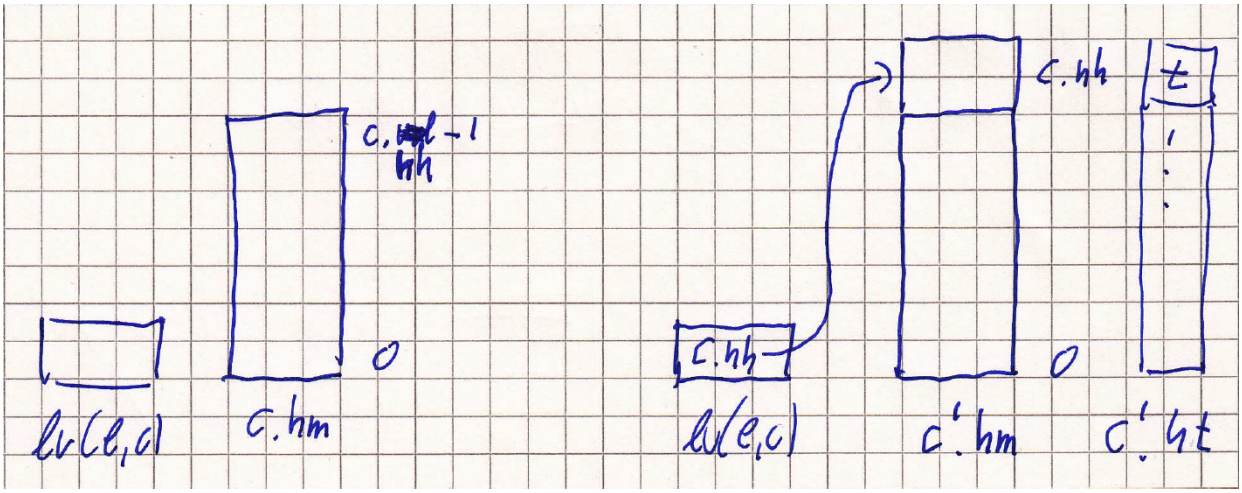


Figure 10.16: A 'new' statement creates a new nameless variable $d.nh-1 = c.nh$ on the heap, records its type t at $d.ht(c.nh)$ and stores $c.nh$ into the pointer variable $lv(e,c)$ designated by e. This makes $lv(e,c)$ point to the newly created variable $c.nh$.

Execution of a 'new' statement does not change recursion depth, stack and result destination stack.

$$X \in \{rd, st, rds\} \rightarrow c'.X = c.X$$

The other components of the configuration change in the following way, which is illustrated in figure 10.16

- the number of heap variables in increased by 1.

  $$c'.nh = c.nh + 1$$

  This creates the new heap variable $c.nh$ and we have

  $$V(c') = V(c) \cup \{c.nh\}$$

- the type of this new heap variable is fixed to t. The type of the previously existing heap variables stays the same

  $$c.ht(x) = \begin{cases} t & x = c.nh \\ c.ht(x) & x < c.nh \end{cases}$$

- the new heap variable is initialized with its default value.

  $$c'.m(c.nh) = deft(t)$$

- the pointer variable $lv(e,c)$ specified by the left hand side is made to point to the newly created variable. The content of all other simple subvariables in $SV(c)$ remain unchanged

  $$\begin{aligned} x \in SV(c) \quad \wedge \quad & simple(vtype(x,c)) \rightarrow \\ c'.m(x) \quad = \quad & \begin{cases} c.nh & x = lv(e,c) \\ c.m(x) & \text{otherwise} \end{cases} \end{aligned}$$

- the new statement is dropped from the program rest

  $$c'.pr = tail(c.pr)$$

One new heap variable $c.nh$ is created. It is initialized with its default value. Default values for subvaribales of the new variable of pointer type are initialized with $null$. Thus invariants $tc(c')$ and $p - targets(c')$ hold trivially for the new variable.

The pointer subvariable $x = lv(e, c)$ has by invariant 7.2 type

$$vtype(x, c) = (vtype(lv(e, c), c) = etype(e, f) = t*$$

and gets assigned the value

$$c'.m(x) = c'.m(lv(e, c')) = c.nh$$

of type

$$vtype(c'.m(x), c') = vtype(c.nh, c') = c'.ht(c.nh) = t$$

Because

$$c'.m(x) \in SV(c') \wedge onheap(c'.m(x), c')$$

we have $tc - p(c')$ and $p - target(c')$ for subvariable x. All other subvariables of c stay unchanged, thus their invariants are maintained. Only the first statement is dropped from the program rest. Thus invariants $inv - pr$ and $inv - rds$ are maintained by the known arguments.

10.6.4 Function Call

If a function g is called, the number p of its parameters is specified in the function table as

$$p = ft(g).p$$

The head of the program rest has the form

$$hd(c.pr) : \begin{cases} e = g(e_1, \ldots, e_p) & p \geq 1 \\ g() & p = 0 \end{cases}$$

The type of function frames for g is specified in the type table table as

$$type(\$g) = \{t_1\ x_1; \ldots; t_s\ n_s\}$$

The type t of the result returned by g is specified in the function table as

$$t = ft(f).t$$

There are several requirements:

Context Condition 19.

- *the expression type of the left hand side is simple and matches the type t of the functions return value*

$$simple(etype(e, f)) \wedge etype(e, f) = t$$

- *for $j \in [1 : p]$ the type of the j'th parameter is simple and matches the type t_j of the j'th parameter x_j in the function declaration.*

$$simple(e_j, f)) \wedge etype(e_j, j)) = t_j$$

The execution of a function call has several effects as illustrated in figure 10.17.

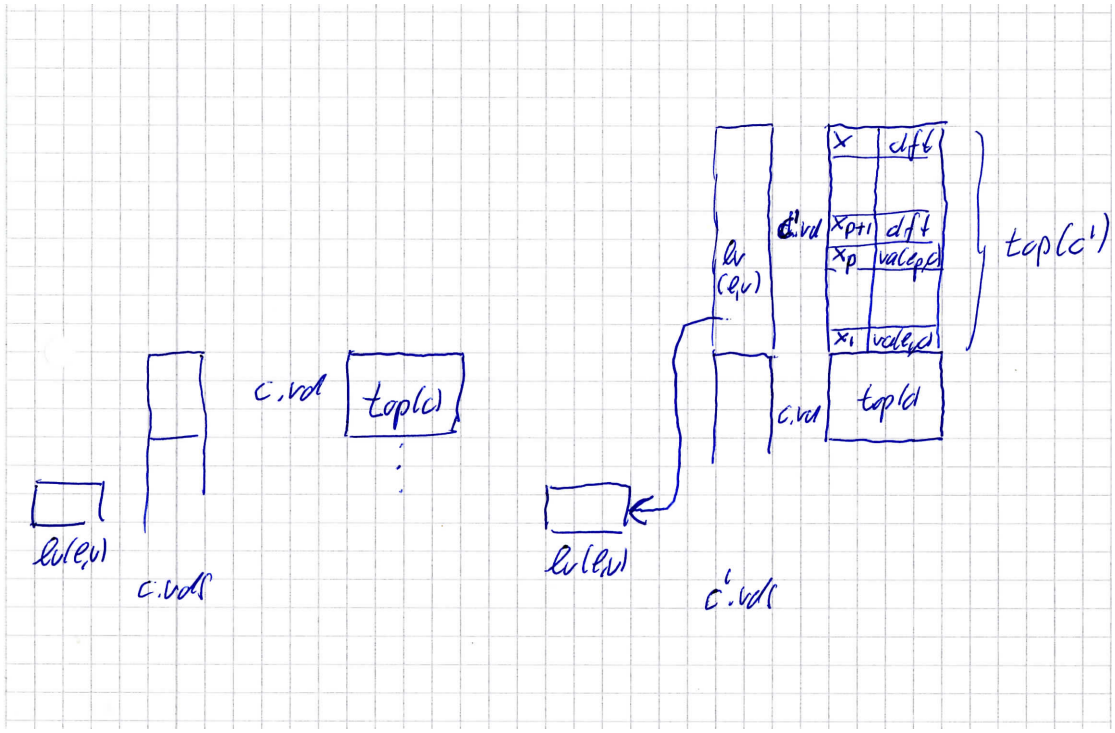


Figure 10.17: A function call increases recursion depth creating a new stack frame $top(c'')$ and a new entry $c'rds(c'.rd)$ on the return destination stack. It passes parameter values $va(e_i, c)$ to paramater x_i of the new top function frame and intializes local variables of the new frame to their default value. A pointer to the result destination $lv(e, c)$ of the call is stored at the new entry $c'.rds(c'.rd)$ of the result destination stack.

- the heap stays the same. The recursion depth is increased to $c.rd+1$ and a new function frame for g is created at the top of the stack

$$\begin{aligned} c'.nh &= c.nh \\ c'.ht &= c.ht \\ c'.rd &= c.rd + 1 \\ c'.st(x) &= \begin{cases} g & x = c'.rd \\ c.st(x) & x \leq c.rd \end{cases} \end{aligned}$$

  Thus the new top frame of the stack is

$$top(c') = (g, c'.rd)$$

  The variables of c' are the old variables together with the new top frame

$$V(c') = V(c) \cup \{top(c')\}$$

- Parameters are initialized with their values

$$j \leq p \rightarrow c'.m(top(c').x_j = va(e_j, c)$$

- local parameters are initialized with their default value

$$j > p \rightarrow c'.m(top(c').x_j = dft(t_j))$$

- the subvariable $lv(e,c)$ where the functions result is returned, is recorded in the new entry $c'.rds(c'.rd)$ of the return destination stack. Old parts of this stack stay unchanged.

$$c'.rds(x) = \begin{cases} lv(e,c) & x = c'.rd \\ c.rds(i) & x \leq c.rd \end{cases}$$

- in the program rest the function call is replaced by the body of g, which is recorded in the function table

$$c'.pr = ft(g).body; \circ tail(c.pr)$$

The new variable $top(c')$ is initialized. This involves two different cases:

- parameters: one argues that $tc - SV(c')$, $tc - p(c')$ and $p - targets(c')$ hold for each of them them as in the case of assignment statements.
- local variables: they are initialized with their default values. One argues that their invariants hold as in the case of the new heap variable in a 'new' statement.

For the new entry $c'.rds(x)$ at argument

$$x = c'.rd$$

of the return destination stack we argue

$$\begin{aligned} vtype(c.rds(x)) &= vtype(lv(e,c)) \\ &= etype(e,f) \quad (\text{part 2 of } inv - expr(e,c)\ ) \\ &= t \quad (\text{context condition}) \\ &= ft(g).t \\ &= ft(c.st(x)).t \end{aligned}$$

This shows part 1 of $inv - rds(c')$. The value $lv(e,c)$ assigned to $c.rds(x)$ by part 1 of invariant $inv - expr(e,c)$ to the old subvariables

$$c'.rds(x) = lv(e,c) \in SV(c)$$

Thus in c' it lies on the heap, in global memory or in the stack under the new to frame. This shows part 2 of $inv - rds(c')$.

Old subvariables are not changed, thus their invariants are preserved. The number of returns in the program rest and recursion depth are both increased by 1. The new front portion of the program rest until the first return from the left (which is the new last return from the right) comes from statements $s \in g$ belonging to the function g of the new top frame, thus invariant $inv - pr$ is preserved.

10.6.5 Return

The head of the program rest has the form

$$hd(c.pr) = return \quad e$$

We require that the type $etype(e,f)$ of the expression in the return statement matches the type t of the return result of the function f containing the return statement

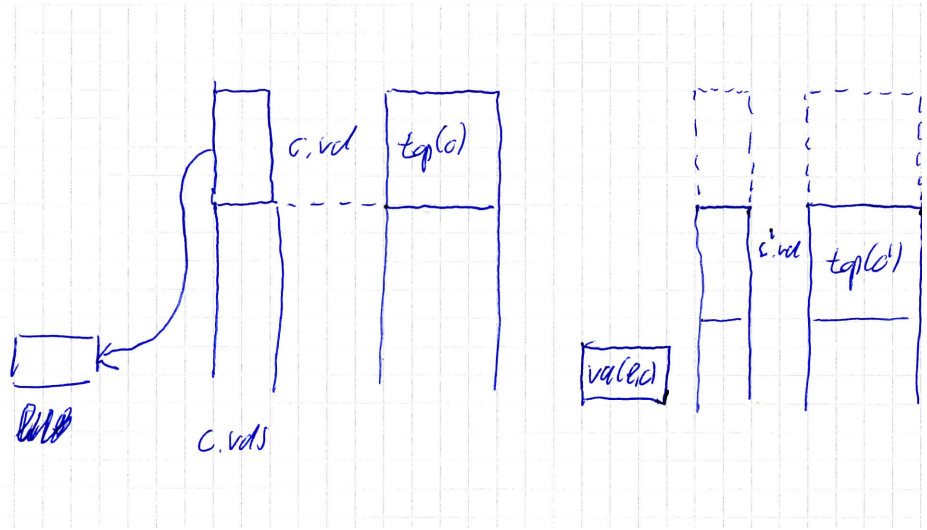


Figure 10.18: A return statement assigns the return value $va(e, c)$ to the subvariable pointed to by the top entry $c.rds(c.rd)$. Then it decreases recursion depth, hereby dropping the old top frame and the old top entry in the return destination stack $c.rds$.

Context Condition 20.

$$t = etype(e, f) = ft(f).t$$

That type t is simple was already required by the context conditions of the function call. Execution of the return statement has the following effects as illustrated in figure 10.18

- the value $va(e, c)$ is assigned to the subvariable

  $$y = c.rds(c.rd),$$

  which is destination of the return result.

  $$c'.m(y)) = va(e, c)$$

  By $inv - rds$ subvariable y has type

  $$vtype(y, c) = t$$

  And thus is y is simple.

- the top frame is deleted from the stack, and the heap is unchanged

  $$\begin{aligned} c'.rd &= c.rd - 1 \\ c'.st(x) &= c.st(x) \text{ for } x \in [0 : c'.rd] \\ c'.nh &= c.nh \\ c'.ht &= c.ht \end{aligned}$$

  Thus

  $$V(c') = V(c) \setminus \{top(c)\}$$

- Subvariables of c other than the return destination do not change

  $$\begin{aligned} x \in SV(c') \wedge simple(x, c) \wedge x \neq y \quad &\rightarrow \\ c'.m(x) &= c.m(x) \end{aligned}$$

- the return statement is dropped from the program rest

$$c'.pr = tail(c.pr)$$

The old top frame disappears. The simple subvariable y is assigned value $va(e, c)$ which has expression type

$$etype(e, c) = t$$

Now $tc(c')$ and -in case t is a pointer type - $p - targets(c')$ are concluded for y as in the case of assignments. The number of returns in the program rest and the recursion depth both decrease by one. The return statement, which is dropped from the program rest was the last statement belonging to the function f of the old top frame. Thus invariant $inv - pr(c')$ follows from $inv - pr(c)$.

10.7 Correctness of $C0$-Programs

We use the C0-semantics to analyze a some small programs. These examples serve several purposes: i) they simply illustrate the constructs of the semantics. ii) they show that properties of C0 computations, i.e. of program runs can be rigorously proven as mathematical theorems. iii) When using computer arithmetic, one *always* has to keep in mind that it is finite. Several examples were chosen to highlight this fact.

10.7.1 Assignment and Conditional Statement

Consider the following program

```
int x;
int main()
{
  x=3;
  x=x+1;
  if (x>0) then {x=1} else {x=2};
  return -1
}
```

It produces a finite computation (c^i). There are no function calls and no new statements. Thus for all c^i we have

$$\begin{aligned} c^i.rd &= 0 \\ c^i.nh &= 0 \\ c^i.st(0) &= main \\ V(c^i) &= \{gm, (main, 0)\} \end{aligned}$$

Function *main* has no parameters and no local variables. A single global variable x is declared. Thus, global memory and function frames for Main have types

$$\begin{aligned} \$gm &= \{int \quad x\} \\ \$main &= \{ \quad \} \end{aligned}$$

As illustrated in figure 10.19 in the body of function *main* variable name x is always bound to

$$lv(x, c^i) = gm.x$$

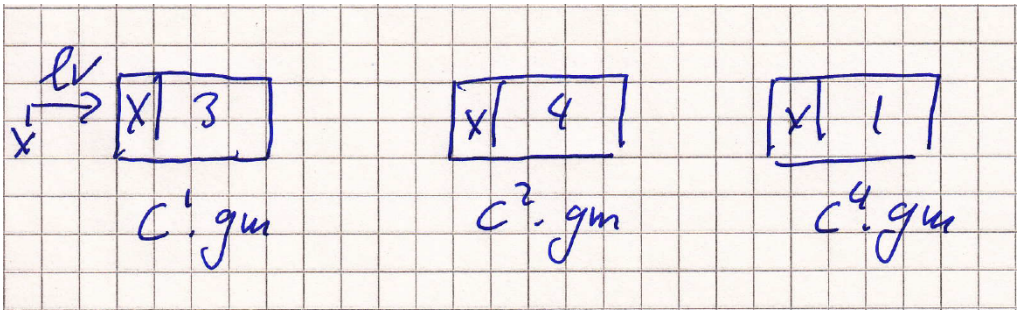


Figure 10.19: Value of $gm.x$ in various configurations

and its value and expression type are

$$\begin{aligned} va(x, c^i) &= c^i.m(gm.x) \\ etype(x, main) &= int \end{aligned}$$

We $va(x, c^i)$ and $c^i.pr$ for $i = 0, \ldots$.

- Initially we have

$$va(x, c^0) = dft(int) = 0 \quad \text{and} \quad c^0.pr = ft(main).body$$

- After execution of the first assignment we have

$$va(x, c^1) = 3$$

The program rest $c^1.pr$ is

```
x=x+1;if (x>0) then {x=1} else {x=2};return -1
```

- After the next step we have

$$va(x, c^2) = 4$$

The program rest $c^2.pr$ is

```
if (x<=0) then {x=1} else {x=2};return -1
```

- Execution of the conditional statement only changes the program rest. Expression $x > 0$ is evaluated to

$$4 > 0$$

which is true, thus The program rest $c^3.pr$ is

```
{x=1};return -1
```

- After the assignment statement we have

$$va(c^4.x) = 1$$

and the program rest is

```
return -1
```

- Execution of the return statement of function main at recursion depth 0 has no semantics yet. In theory we might say the program halts. In practice it should return to the operating system. In later sections we will see how this can be implemented.

10.7.2 Computer Arithmetic

We change in the previous example the first assignment in the body of function *main* to

```
x=2147483647
```

Clearly, in ordinary arithmetic the test $x > 0$ in configuration c^2 would evaluate to true, because

$$2147483647 + 1 > 0$$

However in two's complement arithmetic we get

$$\begin{aligned} va(x, c^1) &= 2147483647 \\ &= 2^{31} - 1 \\ va(x, c^2) &= (2^{31} tmod 2^{32}) \\ &= -2^{31} \\ &< 0 \end{aligned}$$

10.7.3 While Loop

We consider the following example program

```
int n;
int res;
int main()
{
  n=32768;
  res=0;
  while (n>0) do
  {
    res=res+n;
    n=n-1;
  }
  return -1
}
```

We would hope to show that after a certain number of steps T the program ends with a program rest

$$c^t.pr : return \quad -1$$

and for the result holds

$$va(res, c^T) = \sum_{i=1}^{1024} i$$

Variable names n and res are bound in all configurations to global memory componenets

$$\begin{aligned} lv(n, c) &= c.gm.x \\ lv(res, c) &= c.gm.res) \end{aligned}$$

After 2 steps we have

$$\begin{aligned} va(n, c^2) &= 32768 \\ va(res, c^2) &= 0 \end{aligned}$$

and the program rest is

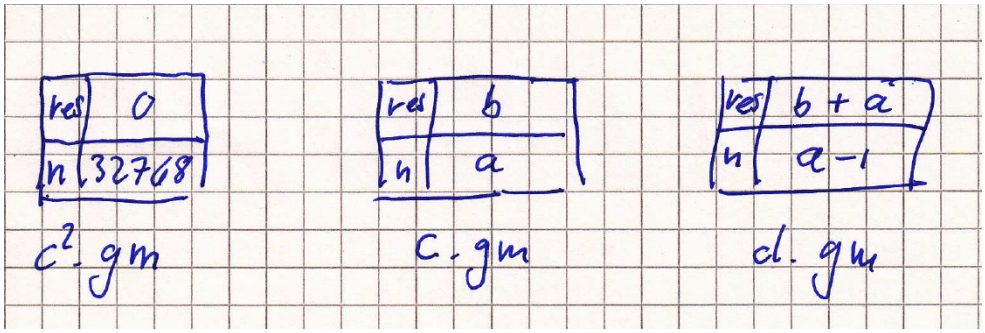


Figure 10.20: Global memory in various configurations of the execution of a while loop

```
while (n>0) do
  {
    result=result+n;
    n=n-1;
  }
  return -1
```

Now let c be a configuration, where 'n is positive'

$$va(n, c) > 0$$

and the program rest is the same as above

$$c.pr = c^2.pr$$

Let

$$d = \delta_C^3(c)$$

the configuration obtained after 3 more $C0$-steps. Analysis along the lines of subsection 10.7.1 gives

$$\begin{aligned} d.pr &= c.pr \\ va(n, d) &= (va(n, c) - 1 \bmod 2^{32}) \\ va(res, d) &= (va(res, c) + va(n, c^t) \bmod 2^{32} \end{aligned}$$

This is illustrated in figure 10.20 By induction on $j \in [0 : 32768]$ one shows

$$\begin{aligned} c^{1+3\cdot j}.pr &= c^1.pr \\ va(n, c^{1+3\cdot j}) &= 1024 - j \\ va(result, c^{1+3\cdot j}) &= \sum_{i=1024-j+1}^{1024} i \end{aligned}$$

The proof is not completely trivial: for the last line one needs to use the formula on arithmetic sums of subsection 3.3.2 to show the absence of overflows:

$$\begin{aligned} \sum_{i=1}^{32768} i &= 1024 \cdot (32768 + 1)/2 \\ &\leq 32768^2 \\ &= 2^{30} \\ &= 2^{31} \end{aligned}$$

For $j = 32768$ we have

$$va(n, c^{1+3 \cdot j} = 0$$

The condition of the while loop is false and with $t = 2 + 3 \cdot 1024$ we get the desired result.

If we would change the initial assignment to n to a number with well known bad properties

```
n = 2147483648
```

we would of course never execute the loop body because

$$\begin{aligned} va(n, c^1) &= (2^{31} t mod 2^{32} \\ &= -2^{31} \\ &= -2^{31} < 0 \end{aligned}$$

10.7.4 Linked Lists

We consider programs with the following declarations of types and global variables

```
typedef LEL* u;
typedef struct{int content, u next} LEL;
```

Type LEL for list element and the type $u = LEL*$ of pointers to list elements were already informally studied in the introductory examples of subsection 10.1.8. Let

$$x[0 : n-1] \in [0 : c.nh - 1]^n$$

be a sequence of n heap variables of configuration c. We say that $x[0 : n-1]$ is a linked list and write

$$llist(x[0 : n-1], c)$$

if the following conditions hold

1. the x_i are distinct
$$i \neq j \rightarrow x_i \neq x_j$$
2. the x_i have type LEL
$$vtype(x_i, c) = c.ht(x_i) = LEL$$
3. for $i < n-1$ the next component of x_i points to x_{i+1} and the next component of x_{n-1} is the null pointer (see figure x)
$$c.m(x_i.next) = \begin{cases} x_{i+1} & i < n-1 \\ null & i = n-1 \end{cases}$$

This is illustrated in figure 10.21.

The following program initializes variable n to a natural number N. We assume $0 < N < 2^{31}$. We show that it creates a linked list of length N.

```
typedef LEL* u;
typedef struct{int content, u next} LEL;
u first;
u last;
int n;
int main()
```

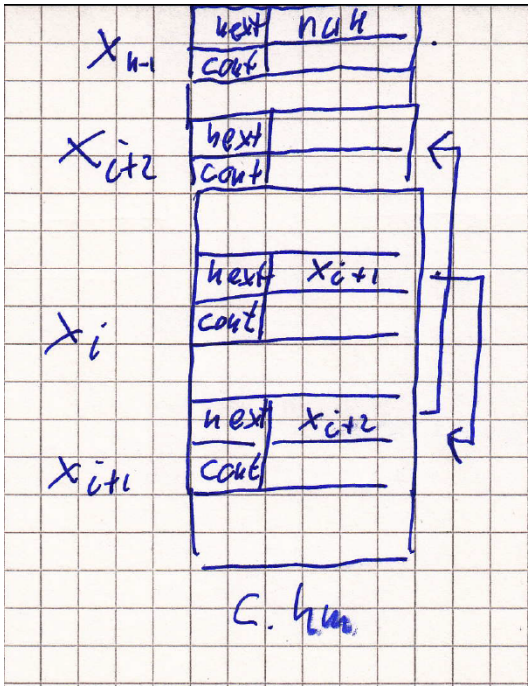


Figure 10.21: Linked list on the heap of a configuration c

```
{
  n=N;
  first=new LEL*;
  last=first;
  n=n-1;
  while N>0
  {
  last*.next= new LEL*;
  last=last*.next;
  n=n-1
  }
  return -1
}
```

After 4 steps of the $C0$ semantics variable n has value $N-1$, there is a single variable of type LEL on the heap; it is initialized with default values. Both pointers point to this variable

$$\begin{aligned} va(^n, c^4) &= n \\ c^4.nh &= 1 \\ c^4.ht(0) &= LEL \\ c.m(0.content) &= 0 \\ c^4.m(0.next) &= null \\ va(first, c^4) &= 0 \\ va(last, c^4) &= 0 \end{aligned}$$

The program rest $c^4.pr$ is

```
while N>0
  {
```

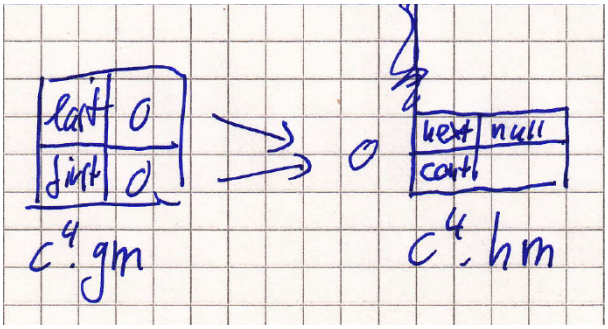


Figure 10.22: In configuration c^4 the heap contains a list of length 1. Pointers $first$ and $last$ point to the single element of this list.

```
last*.next= new LEL*;
last=last*.next
n=n-1
}
return -1
```

As illustrated in figure 10.22 there is a list of length 1 in c with the single element $x_0 = 0$

$$llist(0, c^4)$$

We define the list x of length N by

$$x[0 : N-1] = (0, 1, \ldots, N-1)$$

and prove a lemma that is illustrated in figure 10.23

Lemma 79. *For $i \in [0 : N-1]$ let*

$$c = c^{4+4\cdot i}$$

be the configuration after i iterations of the loop. Then

1. *c has the above program rest ii*

$$c.pr = c^4.pr$$

2. *it has $i+1$ heap variables*

$$c.nh = i + 1$$

3. *$x[0 : i]$ is a linked list in c*

$$llist(x[0 : i], c)$$

4. *pointers $first$ and $last$ point to x_1 and x_i*

$$\begin{aligned} va(first, c^4) &= 0 \\ va(last, c^4) &= i \end{aligned}$$

5. *variable n has value $N - i - 1$*

$$va(n, c) = N - i - 1$$

Proof. For $i = 0$ and $c = c^4$ the statement of the lemma was shown above. The induction step is illustrated in figure **??** Assume the lemma holds for $c = c^{4+3\cdot i}$.

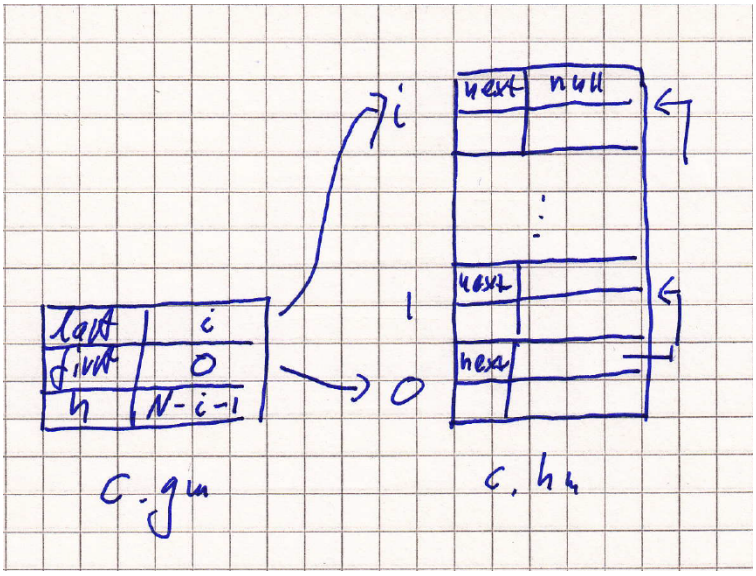


Figure 10.23: In configuration c after i iterations of the loop. The heap contains a list of length $i+1$. Pointers $first$ and $last$ point to the first and last elements of this list.

- The loop condition evaluates to 1 and thus the next program rest $\delta_C(c).pr$ is

  `last*.next= new LEL*; last=last*.next; n=n-1; c.pr`

- Let

  $$c' = \delta_C^2(c)$$

  be the configuration after execution of the new statement. Execution of the new statement increases the number of heap variables

  $$c'.nh = c.nh = i + 2$$

  The new heap variable

  $$x_{i+1} = i + 1 \notin [0 : i]$$

  is distinct from the elements of the existing linked list $x[1 : i + 1]$. It has type LEL and is initialized to its default value

  $$\begin{aligned} vtype(i+1, c') &= LEL \\ va((i+1).content, c') &= c'.m(i+1).content = 0 \\ va((i+1).next) &= c'.m(i+1).next = null \end{aligned}$$

  Applying rules of expression evaluation and applying the induction hypothesis we find that the left hand side of the new statement has left value

  $$\begin{aligned} lv(last*.next, c') &= lv(last*, c').next \\ &= va(last, c').next \\ &= i.next \end{aligned}$$

  To this subvariable of pointer type the new heap variable $i + 1$ is assigned as value and we get

  $$va(i.next, c') = i + 1$$

  Thus we have a linked list of length $i + 1$ in c':

  $$llist(x[1 : i + 1], c')$$

  The program rest $c'.pr$ is

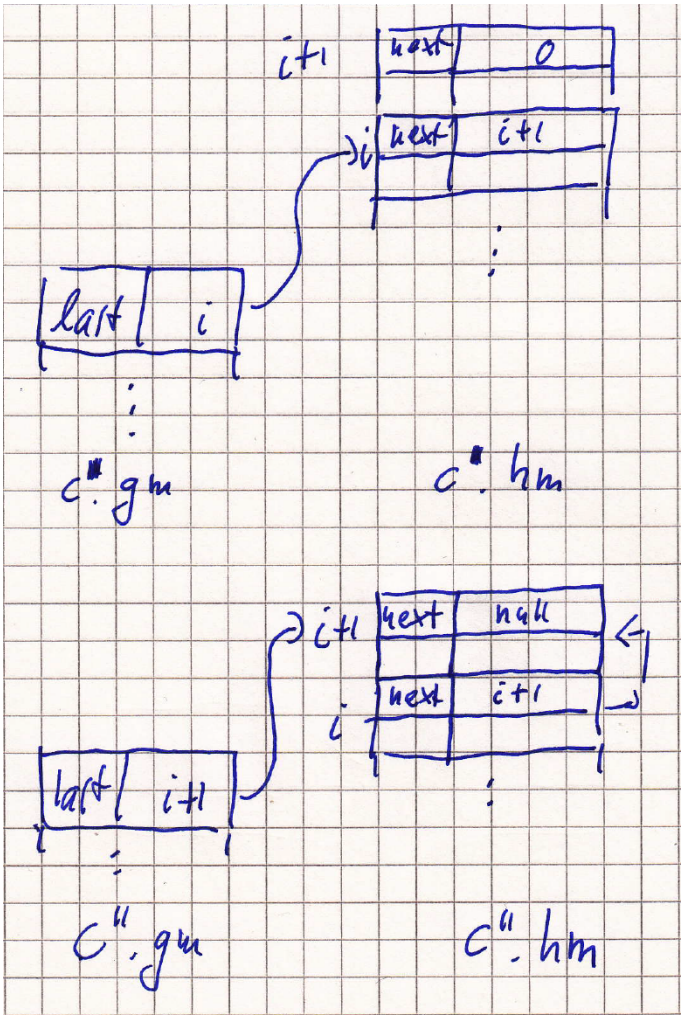


Figure 10.24: In configuration c' the heap contains a list of length $i+2$. In configuration c" *last* points to the last elements of this list.

```
last=last*.next; n=n-1 c.pr
```

- let

$$c" = \delta_C(c') = \delta_C^3(c) = c^{4+3\cdot(i+1)}$$

be the configuration after execution of the assignment statement. Expression evaluation gives

$$\begin{aligned} va(last*.next, c') &= va(last*.c').next \\ &= c.m(lv(last*, c')).next \\ &= c.m(va(last, c')).next \\ &= c.m(i).next \\ &= va(i, c).next \\ &= va(i.next, c) \\ &= i+1 \end{aligned}$$

Execution of the assignment gives

$$va(last, c") = i+1$$

One instruction later n is decremented. Thus the lemma holds for $i+1$

□

For $i = N - 1$ we get

$$llist(x[0 : N - 1], c^{1+3 \cdot i}) \wedge va(n, c^{1+3 \cdot i}) = 0$$

The loop condition evaluates to 0 and the loop is dropped in the next program rest $c^{2+3 \cdot i}.pr$, which is

```
return -1
```

10.7.5 Recursion

Next we analyze a program with recursive function calls. For natural numbers n the Fibonacci numbers $fib(n)$ are defined by

$$\begin{aligned} fib(0) &= 0 \\ fib(1) &= 1 \\ fib(n) &= fib(n-2) + fib(n-1) \end{aligned}$$

A trivial induction gives

$$fib(n) < 2^n$$

This implies that in the calculations below modulo arithmetic coincides with ordinary arithmetic. For the computation of Fibonacci numbers we consider the following program

```
int x;
int Fib(int n)
{
  int res;
  int f1;
  int f2;
  if (n<2) then {res=n} else
  {
    f1=Fib(n-2);
    f2=Fib(n-1);
    res=f1+f2;
  };
  return res
};
```

```
int main()
{
  x=fib(31);
  return -1
}
```

We want to show

$$\exists t : va(x, c^t) = fib(32) \wedge c^t.pr = return \ -1$$

In configurations c for this program, global memory gm and frames for function Fib have types

$$\begin{aligned} \$gm &= \{int \quad x\} \\ \$Fib &= \{int \quad n, int \quad res, int \quad f1, int \quad f2\} \end{aligned}$$

Function $main$ has no local variables or parameters. For $z \in [0 : 31]$ we prove by induction on z the following

Lemma 80. *Let $f \in \{Fib, main\}$. Let*

```
hd(c.pr):  y = Fib(e)
```

be a call of function Fib with parameter e and result destination y. Let

$$z = va(e, c) \leq 31$$

and let

```
hd(c.pr):  y = Fib(e)
```

Then there is a step number t such that for the configuration t steps later

$$c' = \delta^t(c)$$

the following holds

- *only configuration components m and pr changed:*

$$X \notin \{m, pr\} \rightarrow c'.X = c.X$$

- *the function call is dropped from the program rest*

$$c'.pr = tail(c.pr)$$

- *simple variable $lv(y, c)$ is updated with Fibonacci number $fib(va(e, c))$. Other simple variables keep the old value.*

$$x \in SV(c) \wedge simple(x, c) \rightarrow$$
$$c'.m(x) = \begin{cases} fib(va(e, c)) & x = lv(y, c) \\ c.m(x) & otherwise \end{cases}$$

The effect of calling function Fib as specified by the lemma is illustrated in figure 10.25.

Proof. Execution of the function call gets to a configuration d illustrated in figure 10.26 with

$$\begin{aligned}
d.rd &= c.rd + 1 \\
c.st(c.rd) &= Fib \\
cf(d) &= Fib \\
V(d) &= V(c) \cup \{top(d)\} \\
top(d) &= (Fib, d.rd) \\
a \in \{n, f1, f2, res\} \rightarrow lv(a, d) &= top(d).a \\
lv(x, d) &= d.gm.x \\
lv(y, d) &\in SV(c) \\
va(n, d) &= d.m(va(top(d)).n) \\
&= va(e, c) \\
d.rds(d.rd) &= lv(y, c) \\
d.pr &= ft(F).body \circ tail(c.pr)
\end{aligned}$$

The base case of the induction occurs with

$$z = va(n, d) < 2$$

After execution of the condition statement we are in configuration e program rest $e.pr$

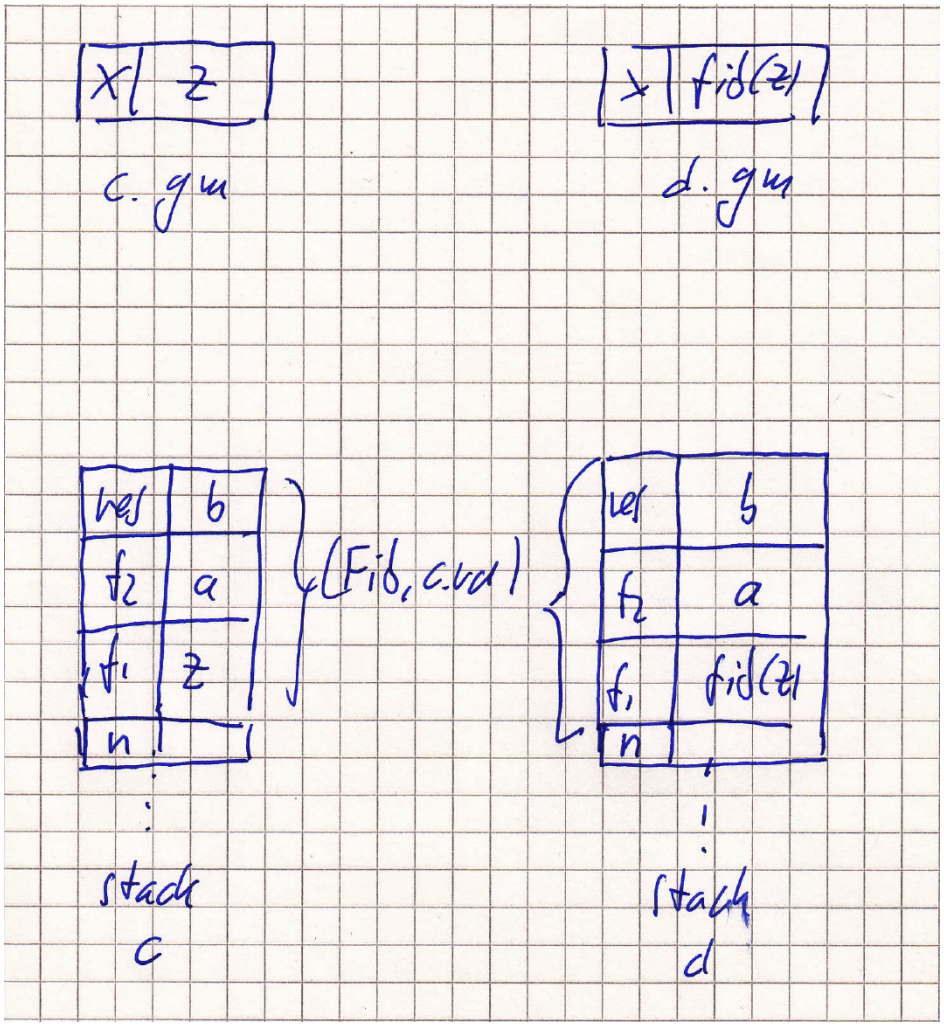


Figure 10.25: Effect of calling function Fib in configuration c. a) If Fib is called from function $main$ with left hand side x, then $gm.x$ is updated. b) If Fib is called recursively from function Fib with left hand side $f1$, then the instance of local variable $f1$ in the top frame is updated.

```
res = n; return res; tail(c.pr)
```

Execution of the assignment leads to configuration f with

$$va(res, f) = n = fib(n)$$

which is illustrated in figure 10.27.

Execution of the return statement gives a configuration h with

$$\begin{aligned}
h.rd &= d.rd - 1 \\
&= c.rd \\
SV(h) &= SV(f) \setminus \{top(d)\} \\
&= SV(c) \\
va(y, h) &= h.m(lv(y, c)) \\
&= h.m(f.rds(f.rd)) \\
&= va(res, h) \\
&= fib(va(e, c)) \\
h.pr &= tail(c.pr)
\end{aligned}$$

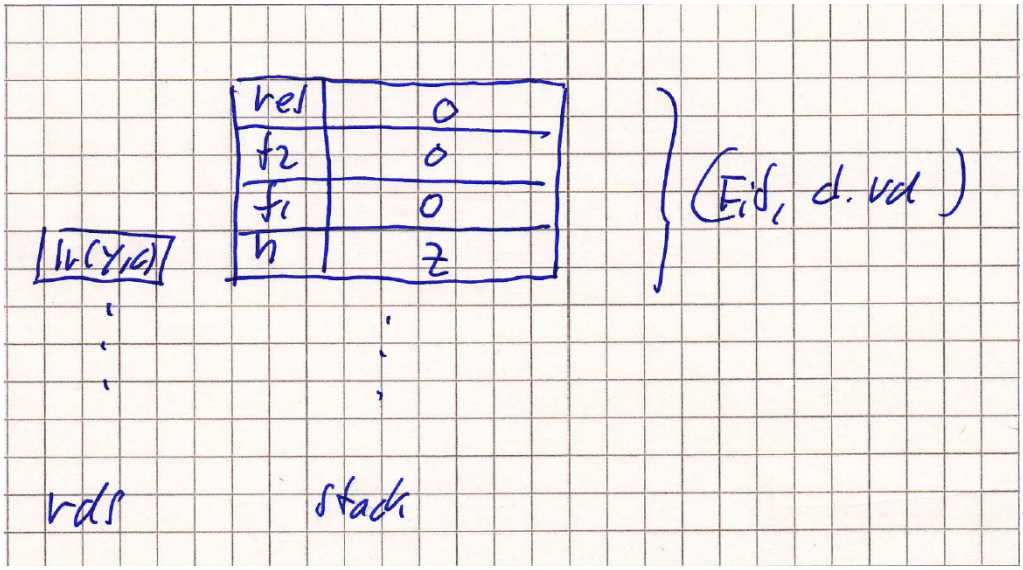


Figure 10.26: Configuration d after the call. The parameter value z is passed to the instance of parameter n in the new top frame. If the call was from function *main* with $y = x$, we have $lv(y, c) = gm.x$. If the call was recursive with $y = f1$ resp. $y = f2$ we have $lv(y, c) = (Fib, c.rd).f1$ resp $lv(y, c) = (Fib, c.rd).f1$, which is in the previous top frame.

This is the situation specified at the right sides of in figure **??**. The case

$$z = va(e, c) \geq 2$$

is handled in the induction step. After execution of the condition statement one gets to a configuration f with program rest $f.pr$:

```
f1=Fib(n-2); f2=Fib(n-1); res=f1+f2;return res; tail(c.pr)
```

The two function calls have parameters with values

$$\begin{aligned} va(n-1, e) &= va(n, c) - 1 \\ va(n-2, c) &= va(n, c) - 2 \end{aligned}$$

Both are smaller than $z = va(n, c)$, thus we can apply the lemma inductively for $z - 1$ and $z - 2$. We conclude for the configuration h after the two calls, which is illustrated in figure 10.28

$$\begin{aligned} va(f1, h) &= fib(va(n, c) - 1) \\ va(f2, h) &= fib(va(n, c) - 2) \end{aligned}$$

The program rest $h.pr$ is

```
res = f1 + f2; return res; tail(c.pr)
```

Execution of the assignment statement gives a configuration u with

$$\begin{aligned} va(res, u) &= va(f1, h) + va(f2, h) \\ &= fib(va(n, c) - 1) + fib(va(n, c) - 2) \\ &= fib(va(n, c)) \end{aligned}$$

For the return statement one argues exactly as in the base case. □

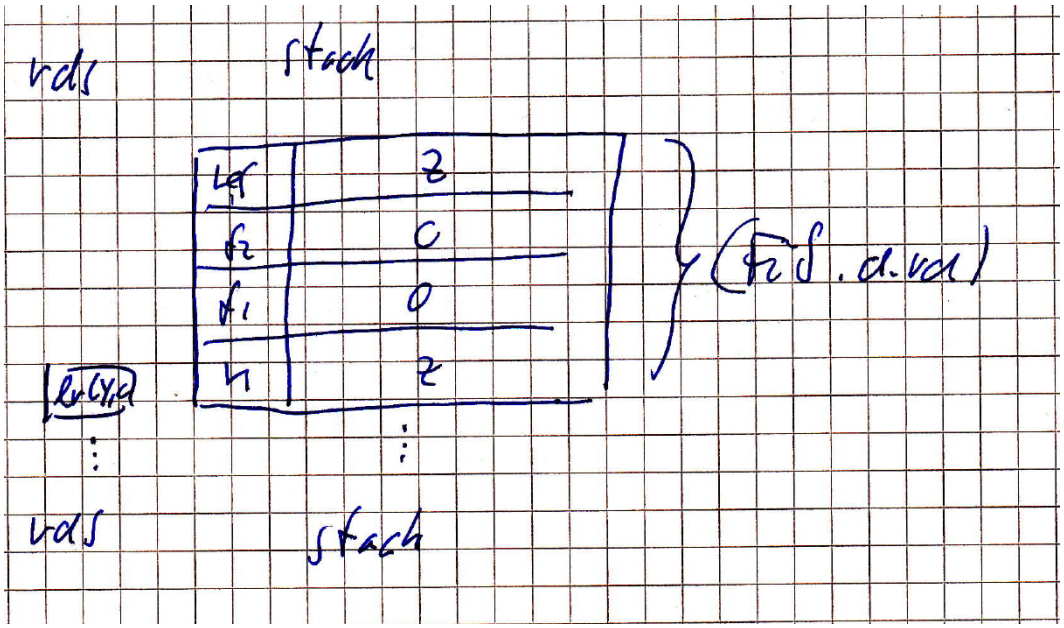


Figure 10.27: Configuration f before the return in the base case . Local variable res in the top frame contains the result $z = fib(z)$. The variable to be updated by the following return statement is $lv(y, c)$ which was stored in the top entry $d.rds(d.rd)$ of the result destination stack

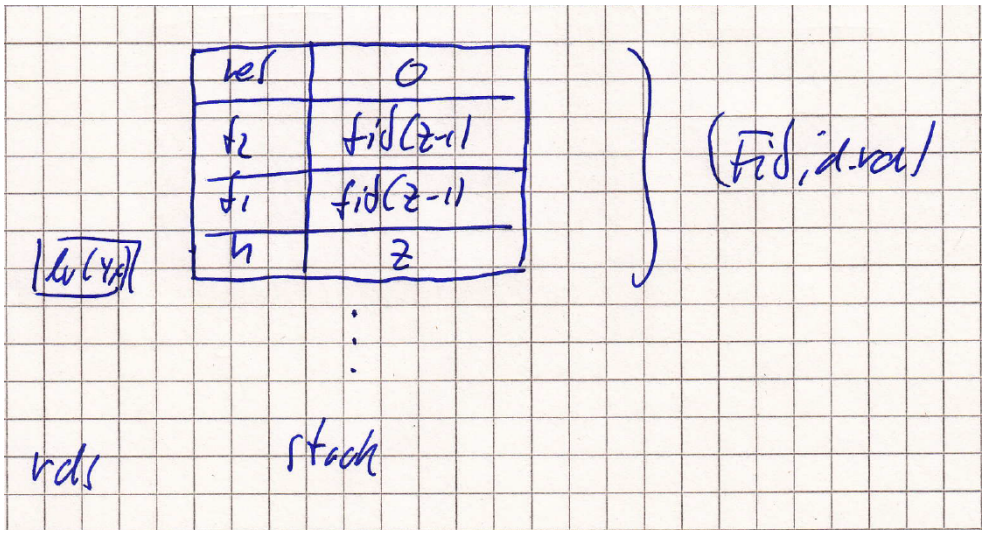


Figure 10.28: Configuration h after 2 recursive calls of function Fib . Local variables $f1$ and $f2$ of the top frame have been updated as illustrated in figure **??** b) with $fib(z-1$ and $fib(z-2)$

Chapter 11

A $C0$-Compiler

Compilers translate source programs p from high level languages L into target programs $code(p)$ from some instruction set architecture ISA, such that the target program simulates the source program. Of course, here we are interested in $C0$ as the source language and $MIPS - ISA$ as the target language. Thus, roughly speaking our $C0$ compiler will compute a translation function function

$$code : L(prog) \to MIPS - ISA$$

In section 11.1 will define a simulation relation

$$consis(c, d)$$

coupling $C0$ configurations c with MIPS configurations d, which will formalize the idea, that the low level MIPS configuration d encodes (more or less) the high level $C0$ configuration c. We are aiming at a step by step simulation of source programs p by target programs $code(p)$. Formally we compare $C0$ computations (c^i) and $MIPS$ computations (d^i) which start in a consistent pair of configurations

$$\begin{array}{rcl} consis(c^0, d^0) & & \\ \wedge \forall i : c^{i+1} & = & \delta_C(c^i) \\ \wedge \forall i : d^{i+1} & = & \delta_M(c^i) \end{array}$$

We then show: if $C0$ configuration c and $MIPS$ configuration d are consistent, and c progresses in one $C0$ step to $c' = \delta_C(c)$, then in some number s of MIPS steps d progresses to a configuration $\delta_M^s(d)$ which is consistent with c'

Lemma 81.

$$consis(c, d) \wedge c' = \delta_C(c) \to \exists s : consis(c', \delta_M^s(d))$$

By induction we get a simulation theorem stating the correctness of non optimizing compilers

Lemma 82. *There exists a sequence $(s(i))$ of steps such that*

$$\forall i : consis(c^i, d^{s(i)})$$

Proof. by trivial induction on i. For the induction step from i to $i + 1$ apply lemma 81 with

$$\begin{array}{rcl} c & = & c^i \\ d & = & d^{s(i)} \end{array}$$

and define

$$s(i + 1) = s(i) + s$$

□

Code generation for expressions is specified in section 11.2 . For expressions we will extend consistency to expressions in a natural way and then generate the obvious code which achieves this. Showing that consistency for expressions is maintained by the generated code will be a bookkeeping exercise. For the order of evaluation of subexpressions we use a simple and elegant algorithm due to Aho and Ullman [**?**], which permits to evaluate expressions with up to 1048567 binary operators using not more than 20 processor registers.

Code generation for statements in section 11.3 will of course aim at maintaining consistency as required in lemma 81 and will also be completely intuitive. The proof of lemma 81, however, will be more than a bookkeeping exercise. This comes from the fact, that code generation is defined by induction over the derivation tree of the program, whereas the program rest is a nice and flat sequence of statements. After the code of a statement $hd(c.pr)$ has run on a $MIPS$ machine (whatever that means), one wishes to show that the program counter $\delta^s(d).pc$ points to the code of $hd(c'.pr)$, but this requires finding (the right instance of) statement $hd(c'.pr)$ in the derivation tree.

In the final section 11.4 of this chapter we study how much of a $C0$ configuration we can reconstruct from a consistent $MIPS$ configuration. *With* the results of this section we will later be able to define the semantics of $C0$ programs with assembly portions in a surprisingly straight forward way.

11.1 Compiler Consistency

11.1.1 Mememory Map

In general a memory map indicates what is stored where in the memory of a processor. We specify here a memory map for processor configurations d which encode $MIPS$ configurations c. Recall that the variables of $C0$ configurations are i) the global memory gm ii) the nameless variables $[0 : c.nh-1]$ on the heap and the stack frames $ST(i,c) = (c.st(i), i)$ for $i \in [0 : c.rd]$. Figure 11.1 shows how these variables and the translated program will be mapped into the memory $d.m$ of $MIPS$ configurations d.

- code: the translated code starts at address a and does not occupy addresses beyond b.

- global memory gm starts at address $sbase$. We maintain a pointer to this address in general purpose register $gpr(28)$.

$$sbase = d.gpr(bpt) \quad \text{with} \quad bpt = 28$$

- the stack frames are stored in order $ST(0,c), ST(1,c) \ldots, top(c)$ starting at the first address behind the space for global memory. During a computation the stack grows an shrinks. It should stay below address $hbase$. We maintain a pointer $sptr$ to the first free location after the top stack frame in register $gpr(29)$.

$$sptr(d) = d.gpr(spt) \quad \text{with} \quad spt = 29$$

- heap memory starts at an address $hbase$ and should not extend beyond address $hmax$. During the computation the heap can only grow (unless we do garbage collection as sketched in section 11.4). We maintain a pointer $hptr$ to the first free location on the heap in register $gpr(30)$.

$$hprt(d) = d.gpr(hpt) \quad \text{with} \quad spt = 30$$

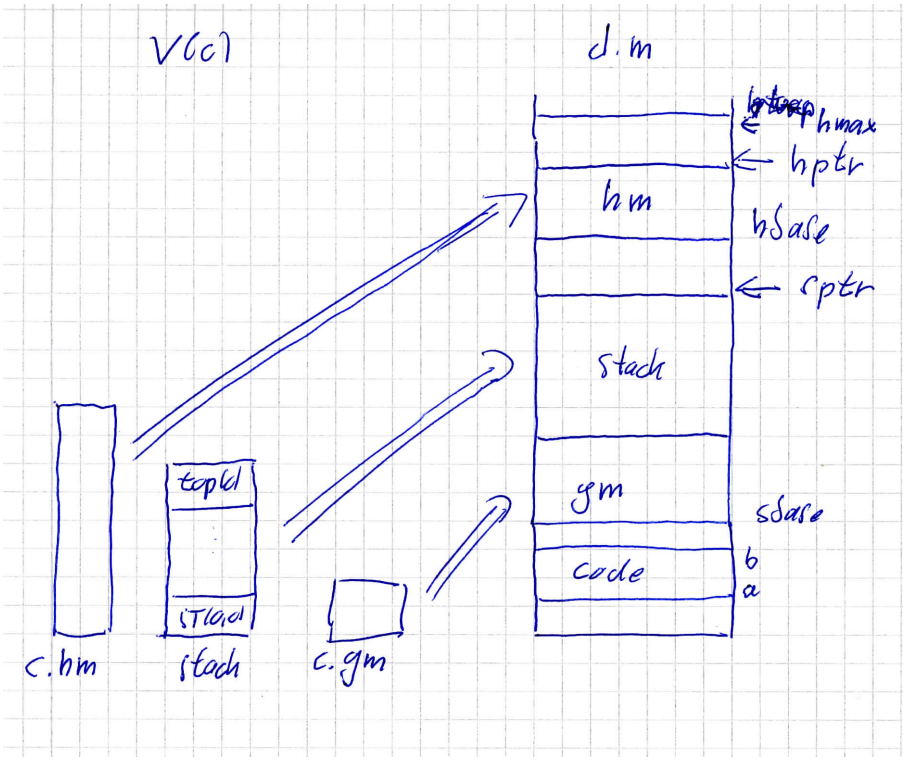


Figure 11.1: Memory map for the compiler. Code is stored between addresses a and b. Global memory starts at address *sbase*. The stack is stored on top of global memory. The stack starts at address *hbase*. Pointers *sptr* and *hprtr* are maintained to the first free addresses after the occupied portions of the stack resp. the heap.

11.1.2 Size of Types, Displacement and Base Address

We compute for each type $t \in ET \cup TN$ the number of bytes $size(t)$ allocated to store a value from the range $ra(t)$ of type t. The definition is straight forward: for values with simple types we will allow 4 bytes. The size of composite types is the sum of the sizes of their components and .... this is not all: frames for functions f without parameters and local variables have an empty struct type

$$\$f = \{\quad\}$$

with no components. This type has size 0.

$$size(t) = \begin{cases} 0 & t = \{\quad\} \\ 4 & simple(t) \\ n \cdot size(t') & t = t'[n] \\ \sum_{i=1}^{s} size(t_i) & t = \{n_1 \quad t_1, \ldots t_s \quad n_s\} \end{cases}$$

With the sizes we can define for each $C0$-variable $x \in V(c)$ the base address $ba(x, c)$ in the MIPS memory where x is stored.

- Global memory starts at address *sbase*.

$$ba(gm, c) = sbase$$

- Add to this the size of global memory + 8and you have the base address of stack frame $ST(0, c)$. Increase the base address of $ST(i, c)$ by 8 plus the size $size(\$c.st(i))$ of the stack frame and

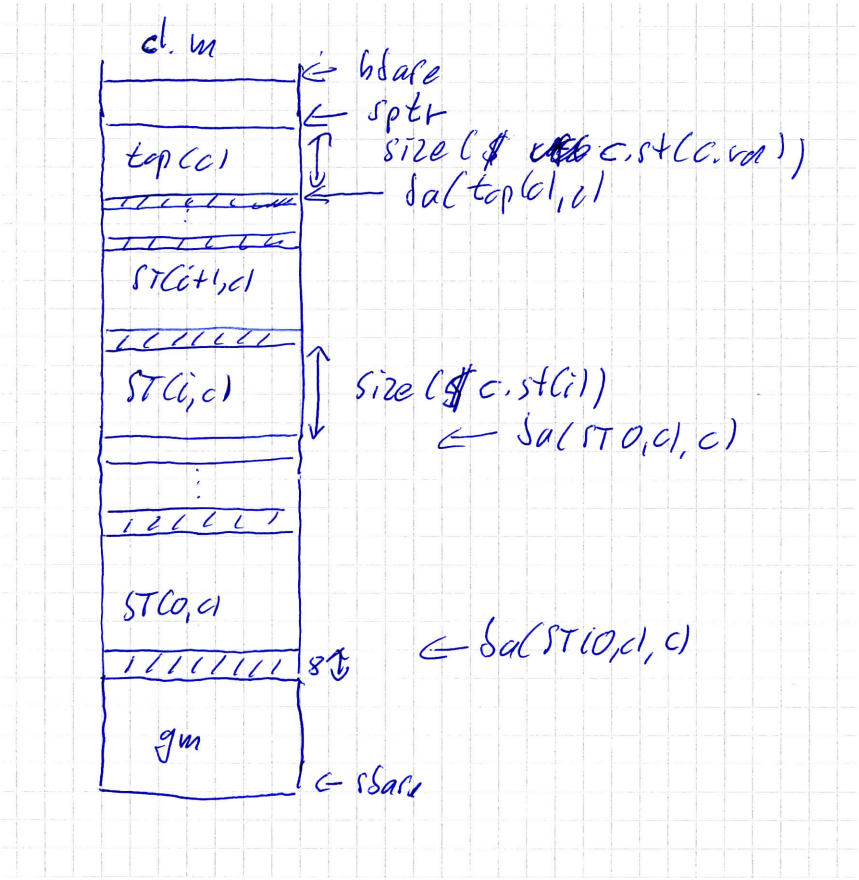


Figure 11.2: Memory map for the stack. Stack frames $ST(i, c)$ are stored in the order of their indices in top of the global memory frame $c.gm$. Below the base address of each frame 2 words = 8 bytes are reserved for auxiliary data.

obtain the base address of the next stack frame. At the two words *below* the base address of each stack, we reserve room for auxiliary data for the frame; this will be the return result destination and the address where we jump when the function returns.

$$ba(ST(i,c)) = \begin{cases} sbase + 8 + size(\$gm) & x = ST(0,c) \\ ba(ST(i,c) + 8 + size(\$c.st(i)) & x = ST(i+1,c) \end{cases}$$

The resulting details of the memory map are shown in figure 11.2

- Nameless heap variable 0 starts at address *hbase*. Increase the base address of heap variable i by the size $size(c.ht(i))$ of this variable and obtain the base address of the next heap variable

$$ba(i,c) = \begin{cases} hbase & i = 0 \\ ba(i,c) + size(c.ht(i))) & x = i + 1 \end{cases}$$

The resulting details of the memory map are shown in figure 11.3

So far our definitions are illustrated by figure x. For the base addresses of subvariables, we zoom into the composite variables and subvariables as shown in figure y. Assume x is a composite variable of type

$$vtype(x) = t$$

with base address $ba(x, c)$ and components $x[i]$ if $t = t'[n]$ is an array type and $t.n_i$ if $t = \{n_1 \quad t_1, \ldots t_s \quad n_s\}$ In variables x of type t we store components $x[i]$ or $x.n_i$ successively starting at low addresses as

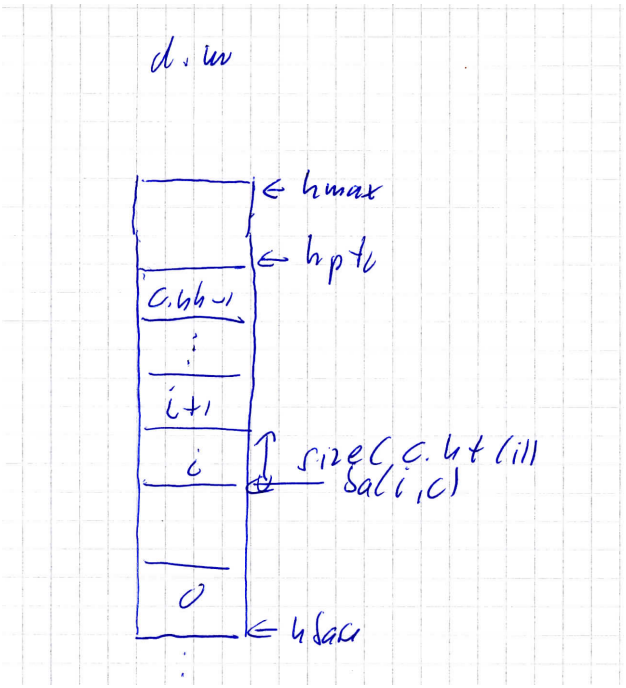


Figure 11.3: Memory map for the heap. Nameless variables $i \in [0 : c.nh - 1]$ are stored in the order of their indices in top of address *hbase*.

shown in figure y. The distance where component $x[i]$ resp. $x.n_i$ starts relative to the base address $ba(x, c)$ is called the displacement $displ(i, t)$ resp. $displ(n_i, t)$. It equals the sum of the sizes of the components to the right. For the two cases we define the displacements as

$$\begin{aligned} displ(i,t) &= i \cdot size(t') \quad \text{if} \quad t = t'[n] \\ displ(n_i,t) &= \sum_{j<i} size(t_i) \quad \text{if} \quad t = \{n_1 \quad t_1, \ldots t_s \quad n_s\} \end{aligned}$$

and the base addresses of the components as

$$\begin{aligned} ba(x[i],c) &= ba(x,c) + displ(i,t) \quad \text{if} \quad t = t'[n] \\ ba(x.n_i,t) &= ba(x,c) + displ(n_i,t) \quad \text{if} \quad t = \{n_1 \quad t_1, \ldots t_s \quad n_s\} \end{aligned}$$

We extend interval notation $[a : b]$ to bit strings $a, b \in \mathbb{B}^{32}$ by defining

$$\begin{aligned} [a:a] &= \{a\} \\ [a:b +_{32} 1_{32} &= [a:b] \cup \{b +_{32} 1_{32}\} \end{aligned}$$

and for subvariables x we define the address range $ar(x, c)$ as the set of addresses allocated to subvariable x:

$$ar(x,c) = [ba(x,c) : ba(x,x) +_{32} size(vtype(x,c))]$$

Note that the address range of simple variable x are the 4 byte addresses of the memory word starting at the base address of the subvariable.

$$simple(x,c) \rightarrow ar(x,c) = [ba(x,c) : ba(x,x) +_{32} 4$$

From the definitions we infer that the address ranges of variables are disjoint

$$x, y \in V(c) \rightarrow ar(x,c) \cap ar(y,c) = \emptyset$$

For a subvaribale $x \in SV(c)$ with array type $t = t'[n]$ the address ranges of subvariables $x[i]$ are disjoint and lie all in the address range of x.

$$\begin{aligned} vtype(x) = t'[n] \wedge 0 \leq i < j < n &\rightarrow \\ ar(x[i], c) \cap ar(x[i], c) &= \emptyset \\ \wedge ar(x[i], c) &\subseteq ar(x, c) \end{aligned}$$

A similar statement holds if x has struct type:

$$\begin{aligned} vtype(x) = \{n_1 \quad t_1, \ldots t_s \quad n_s\} \wedge 1 \leq i < j \leq s &\rightarrow \\ ar(x.n_i, c) \cap ar(x.n_j, c) &= \emptyset \\ \wedge ar(x.n_i, c) &\subseteq ar(x, c) \end{aligned}$$

For subvariables $x, y \in SV(c)$ we say that x is a subvariable of y if $x = ys$ for some selector sequence $s \in S^+$

$$subvar(x, y) \equiv \exists s \in S^+ : x = ys$$

An easy induction shows for subvariables $x, y \in SV(c)$ shows: i) if x, is a subvariable of y, then the address range of x is contained in the address range of y. If neither x is a subvariable of y nor y is a subvariable of x, then their address ranges are disjoint.

Lemma 83. *Let $x, y \in SV(c)$. Then*

$$\begin{aligned} subvar(x, y) &\rightarrow ar(x, c) \subseteq ar(y, c) \\ \sim subvar(x, y) \wedge \sim subvar(y, x) &\rightarrow ar(x, c) \cap ar(y, c) = \emptyset \end{aligned}$$

11.1.3 Consistency for Data, Pointers and Stack

For elementary data types $t \in ET$ we define an encoding $enc(va) \in \mathbb{B}^{32}$ encode values $va \in ra(t)$ in the range or t into $\mathbb{B}^{32}$ in an obvious way. Integers are encoded as two's complement numbers, unsigned integers are encoded as binary numbers. Boolean values are zero extended. Character constants are coded in ASCII code and then zero extended

$$enc(va) = \begin{cases} twoc_{32}(va) & t = int \\ bin_{32}(va) & t = uint \\ 0^{31}va & t = bool \\ 0^{24}ascii(va) & t = char \end{cases}$$

Let c be a $C0$ configuration and let d be a $MIPS$ configuration.

- We say that configurations c and d are consistent with respect to elementary subvariables and write $e - consis(c, d)$ if for all subvariables x with an elementary data type $vtype(x, c) \in ET$ the obvious encoding of its value $c.m(x)$ is stored in the MIPS memory $d.m$ at the address range of x (see figure 11.4).

$$vtype(x) \in ET \rightarrow enc(c.m(x)) = d.m_4(ba(x, c))$$

- We say that configurations c and d are consistent with respect to pointers and write $p - consis(c, d)$ if for all subvariables u with $pointer(u, c)$ the following holds: if u points in configuration c to subvariable v, i.e. $c.m(u) = v$, then the base address $ba(v, c)$ is stored in the MIPS memory $d.m$ at the address range of u (see figure 11.5)

$$(pointer(u, c) \wedge c.m(u) = v \rightarrow d.m_4(ba(v, c)) = ba(u, v))$$

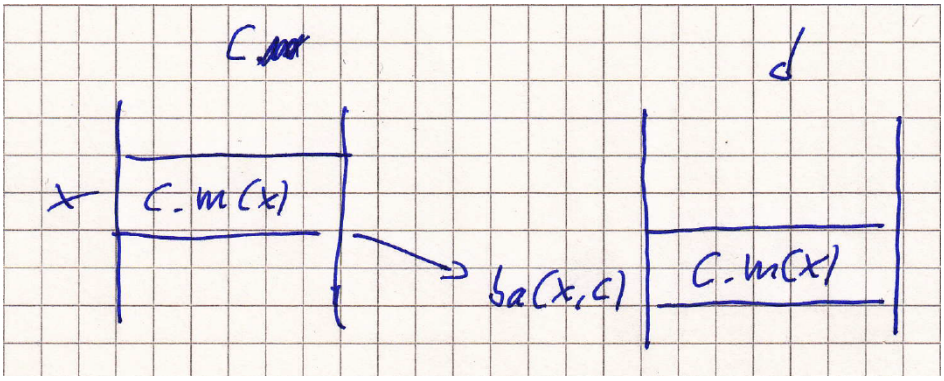


Figure 11.4: Illustration of $e - consistenc(c, d)$. The value $c.m(x)$ of elementary variables x is stored in $d.m$ at the memory word starting at the base address $ba(x, c)$

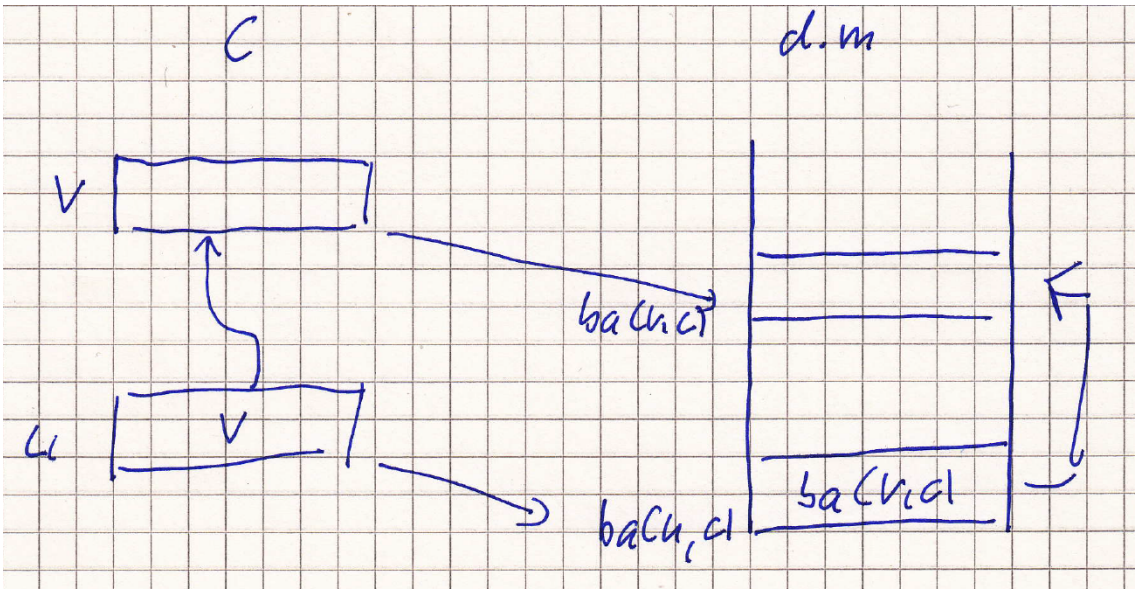


Figure 11.5: Illustration of $p - concis(c, d)$. Pointer u points in c to subvariable v. Then the base address $ba(v, c)$ is stored in $d.m$ at the memory word starting at the base address $ba(u, c)$

- We say that c and d are consistent with respect to base, stack and heap pointers and write $bsp - consis(c, d)$ if i) the pointers point to $sbase$, the first free address on the stack and the first free address on the heap. Moreover ii) the pointers should not be out of range (see figures 11.2 and 11.3)

$$\begin{aligned} d.gpr(bpt) &= sbase \\ d.gpr(spt) &= ba(top(c), c) + size(\$cf(c)) < hbase \\ d.gpr(hpt) &= ba(c.nh - 1) + size(c.ht(c.nh - 1) \leq hmax \end{aligned}$$

- Entries $c.rds(i)$ of the return destination stack are treated like pointers which are stored in the memory word below the base address $ba(ST(i, c),)$ of stack frame $ST(i, c)$. I.e. if $c.rds(i) = u$ then the base address $ba(u, c)$ of u is stored in the MIPS memory at $d.m_4(ba(ST(i, c),) -_{32} 4_{32}$ (see figure 11.6). Thus we say that c and d are consistent with respect to the return destination stack and write $rds - consis(c, d)$ if

$$c.rds(i) = u \rightarrow d.m_4(ba(ST(i, c),) -_{32} 4_{32} = ba(c, u)$$

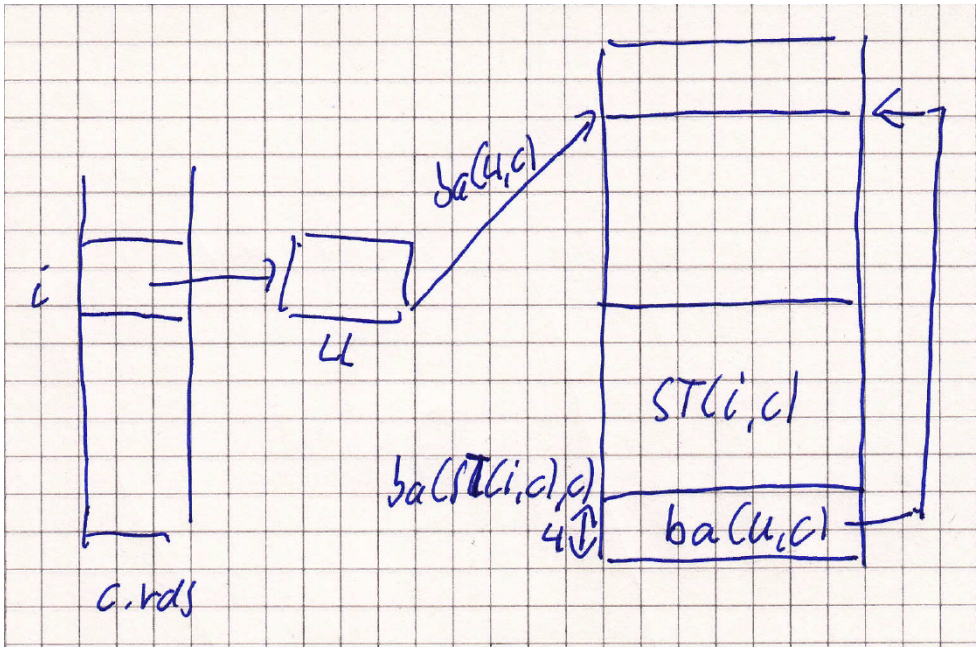


Figure 11.6: Illustration of $rds - concis(c,d)$. Pointer $c.rds(i)$ points in c to subvariable u. Then the base address $ba(,c)$ is stored in $d.m$ at the memory word below the base address $ba(ST(i,c),c)$ of stack frame $ST(i,c)$

11.1.4 Consistency for Code

We simply want to formalize, that the target program of the compilation process is stored in the address range $[a:b]$ and that the code is not changed while to code is running. Code generation will proceed recursively (from sons to fathers) along the nodes n of the derivation tree T of the program, that is translated. For such nodes n we define $code(n)$ as the code generated for the subtree with node n (see figure 11.7). We define its length (measured in bytes) by $|code(n)|$.

$$code(n) \in \mathbb{B}^{8\cdot|code(n)|}$$

This code occupies in the initial configuration d^0 of the MIPS computation $|code(n)|$ addresses fromm address $start(code(n))$ to address $end(code(n))$

$$\begin{aligned} d^0.m_{|code(n)|}(start(code(n)) &= code(n) \\ end(code(n)) &= start(code(n)) +_{32} (|code(n)| - 1)_{32} \end{aligned}$$

Let

$$FN = \{f_1, \ldots, f_k\}$$

be the names of the declared functions in the order they are declared functions in the program and recall, that for $i \in [1:k)$ we defined in subsection 10.2.3 the node

$nbody(f_i)$ as the node in the derivation tree, from which the body of function f_i is derived. For the root ϵ of the derivation tree we get the target program tp as the code of the translated source program as

$$tp = code(\epsilon) = code(nbody(f_1)); \ldots; codenbody(f_k)$$

We say that c and d are code consistent and write $code - consis(c,d)$ if the following conditions hold

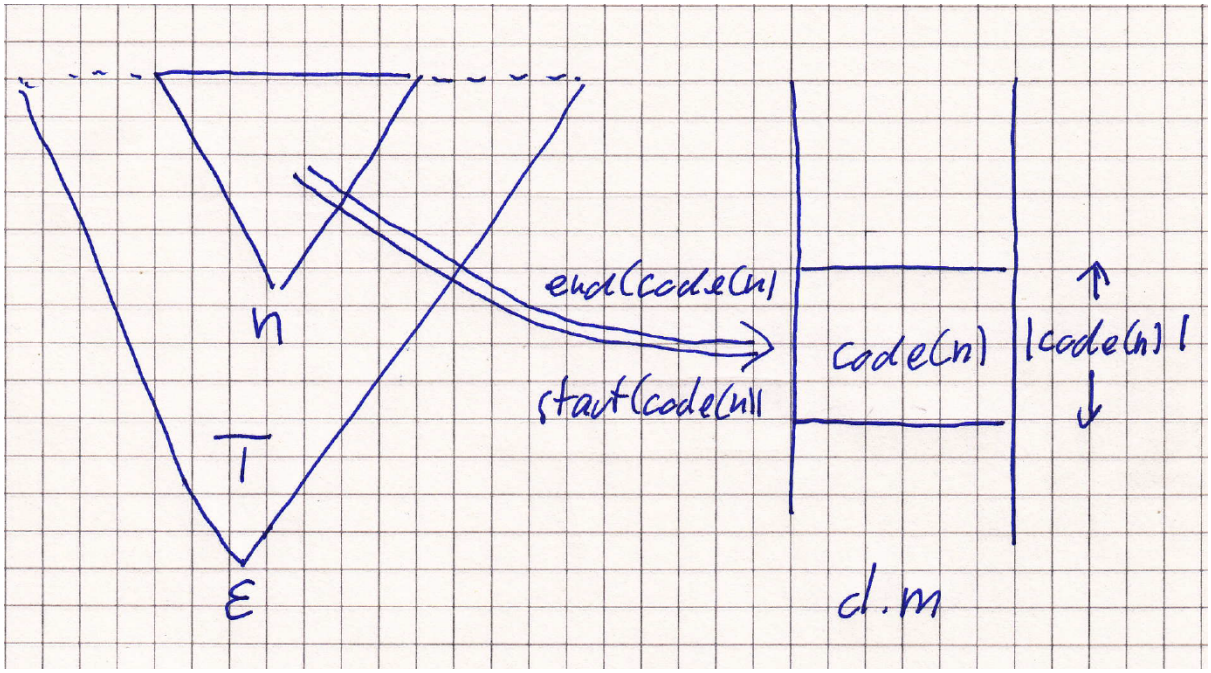


Figure 11.7: The code $code(n)$ generated for node n in the derivation tree is stored at $|code(n)|$ consecutive addresses in $d.m$ from $start(code(n))$ to $end(code(n))$.

- in the initial configuration d^0 the target program tp lies in the code region $[a : b]$

$$\begin{aligned} quad[start(tp) : end(tp)] &\subseteq [a : b] \\ d^0.m_{|tp|}(start(tp)) &= tp \end{aligned}$$

- the target program did not modify itself and is still there

$$d.m_{|tp|}(start(tp)) = tp$$

Note that defining $code(s)$ for statements s in the source program or $code(e)$ for expressions will in general *not* be well defined. We might have *instances* of the assignment $x = x+1$ in two functions of the program: in one function f with local variable x and in another function f' without local variable or parameter x. The statements are executed in configurations c resp c' with $cf(c) = f_i$ and $cf(c') = f_j$. In these configurations the binding

$$\begin{aligned} lv(x, c) &= top(c).x \\ lv(x, c') &= gm.x \end{aligned}$$

are computed in a different way, and this will be reflected in the compilation. Thus $code(x+1)$ would be ambiguous. However, if the two instances of assignment $x + 1$ are border words of nodes n and n' as shown in figure 11.8 , then $code(n)$ refers to the code for the instance of the statement in the body of f and $code(n')$ to the instance in the body of f'.

11.1.5 Consistency for the PC and the Caller Stack

Before we define consistency for the program counter we introduce two so called *ghost* components to $C0$ configurations. They are not needed to define semantics; we obviously have done it without them. They are also not needed to define the compilation process. We only will need them to prove that the compilation is correct, i.e. to establish a simulation theorem between runs of source and

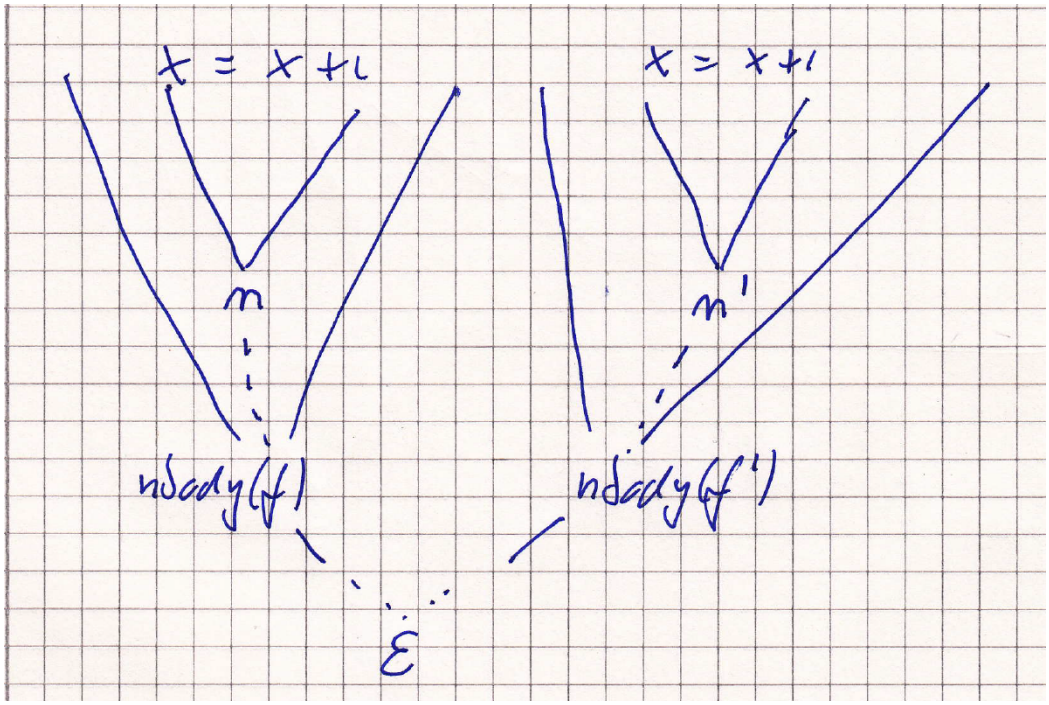


Figure 11.8: Different instances of statement $x = x + 1$ are generated in the bodies of functions f and f'. The instances will be translated differently into $code(n)$ and $code(n')$.

target program. Intuitively we want to define that c and d are pc-consistent, if the program counter $d.pc$ points to the start of the code of the first statement of the program rest

$$d.pc = start(code(hd(c.pr)))$$

Unfortunately, components $c.pr[i]$ are statements; we have seen that different instances of the same statement can exist in the same program, and thus we have to disambiguate $code(c.pr[i])$ by specifying the nodes

$$c.prn[i] \in T.V$$

which identify for each i the instance of $c.pr[i]$ which is meant and to whose code the program counter should point. Thus we will define a ghost component

- $$c.prn \in T.V^+$$

of $C0$ configurations c with these nodes and we will call c and d pc-consistent and write $pc - consis(c, d)$ if the program counter points to the start of the code generated by the subtree with root $hd(c.prn)$ (see figure 11.9)

$$d.pc = start(code(hd(c.prn))$$

For the formal definition the definition of this *program rest nodes* component we proceed in the obvious way: we translate the definition of the program rest $c'.pr$ for the next configuration c' after c into a definition for $c'.prn$ which uses nodes in the derivation tree $T = (V, \ell)$ instead of statements. Details are slightly technical.

Before we do this we remind the reader of a concept from subsection 10.1.9. Let $n \in V$ be a node labeled $\ell(n) = StS$ and with border word

$$bw(n) = s(i1); \ldots; s[m] \quad \text{with} \quad s(i) \in L(St)$$

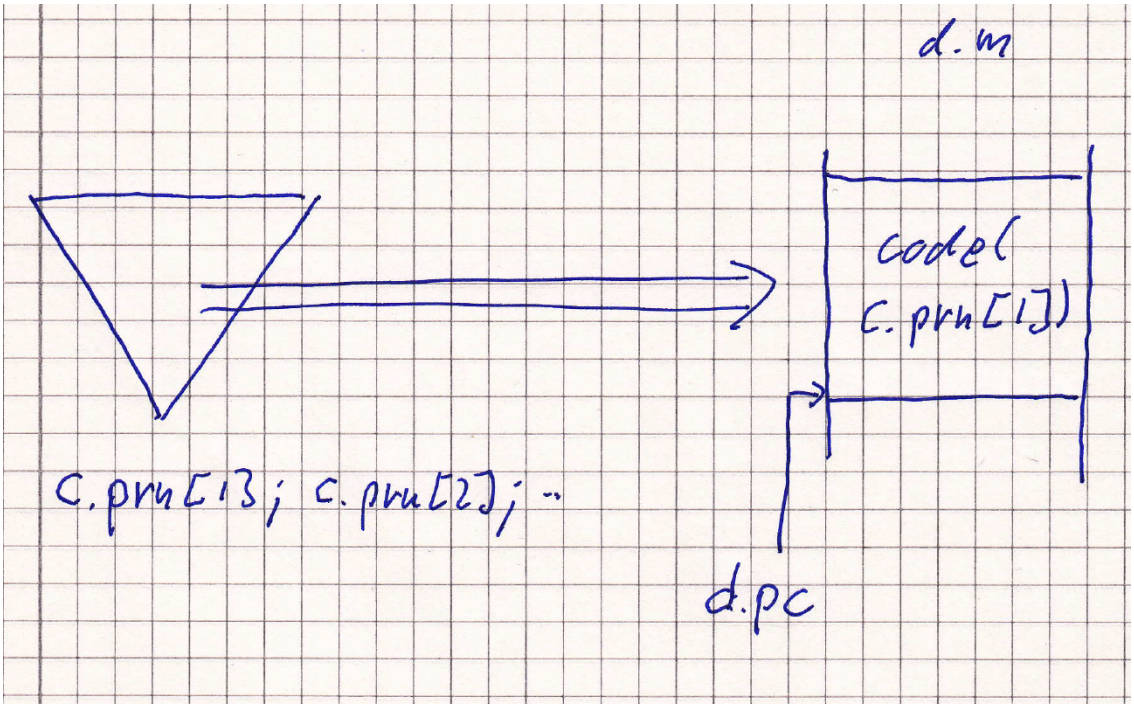


Figure 11.9: Illustration of $pc - consis$. The pc points to start of the code generated for the subtree with root $c.prn[1]$

Let $x[1:m] \in V^m$ be the sequence of descendants of n labeled with $\ell(x_i) = St$ such that we have for all i

$$s(i) = bw(x[i])$$

Thus we have

$$bw(n) = bw(x[1]); \ldots; bw(x[m])$$

We have seen, that we get from n in the tree to $x[i]$ by following $i-1$ times label 2 and then label 0 (see figure 11.10)

$$x[i] = n2^{i-1}0$$

and we called the sequence $x[1:m]$ of these nodes the flattened sequence of n

$$fseq(n) = x[1:m]$$

Now consider the body of function f as shown in figure 11.11

$$ft(f).body = s(1); \ldots s(m); returne$$

It is derived from node $n = nbody(f)$ labeled $body$. Following edge 0 we arrive at a node deriving the statement sequence $s[1:m]$ and following edge 2 we arrive at a node deriving the return statement. Thus we define the sequence of statement nodes of the body of f as

$$snodes(f) = fseq(nbody(f)0); nbody(f)2$$

initial configuration. For the initial configuration c^0 we define $c^0.prn$ as the sequence of statement nodes in the body of function $main$

$$c^0.prn = snodes(main)$$

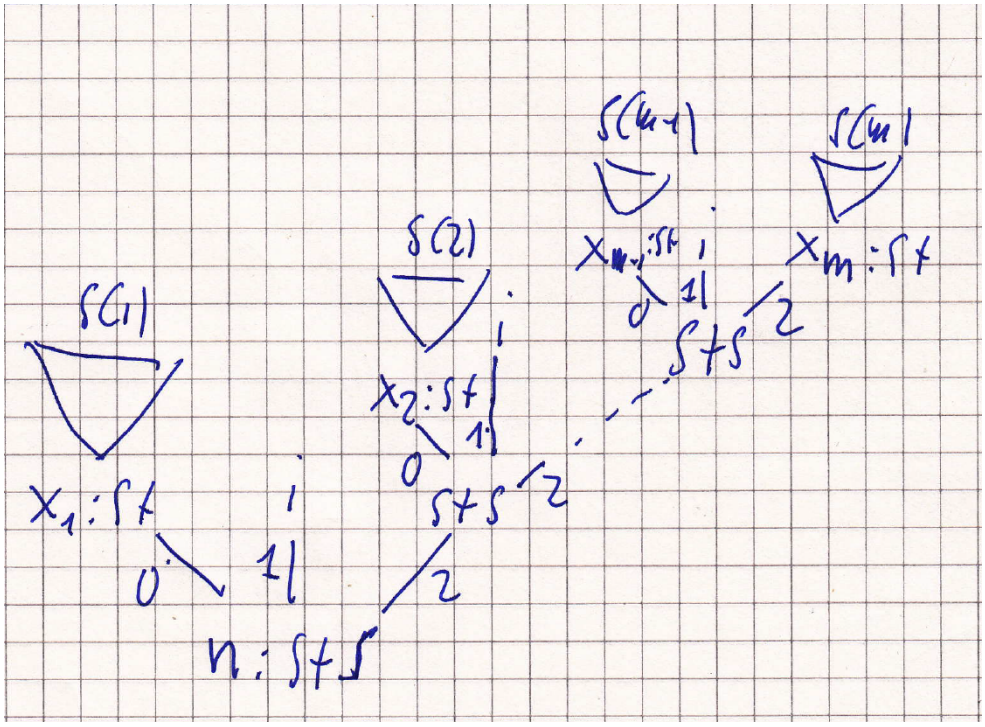


Figure 11.10: If n is labeled StS then descendant x_m with label St is reached following 2^{i-1} times label 2 and then label 0. For each i the border word of x_i is statement $s(i)$.

Assignments, new statements and return statements. If $hd(c.pr$ is an assignment statement, a new statement or a return statement, then the first node in $c.prn$ is dropped

$$c'.prn = tail(c.prn)$$

While loops. Let

`hd(c.pr): while e { body }`

As shown in figure 11.12 this statement is derived by node $n = hd(c.prn)$ labeled St; following edge 3 we arrive at a node labeled StS which derives the loop body. We define

$$c'.prn = \begin{cases} fseq(hd(c.prn)3); c.prn & va(e,c) = 1 \\ tail(c.prn) & va(e,c) = 0 \end{cases}$$

If then else. Let

`hd(c.pr): if e then {if-part} else {else-part}`

As shown in figure 11.13 this statement are derived by node $n = hd(c.prn)$ labeled St; following edge 5 we arrive at a node labeled StS which derives the if-part and following edge 9 we arrive at a node labeled StS which derives the else- part. We define

$$c'.prn = \begin{cases} fseq(hd(c.prn)4); tail(c.prn) & va(e,c) = 1 \\ fseq(hd(c.prn)7; tail(c.prn) & va(e,c) = 0 \end{cases}$$

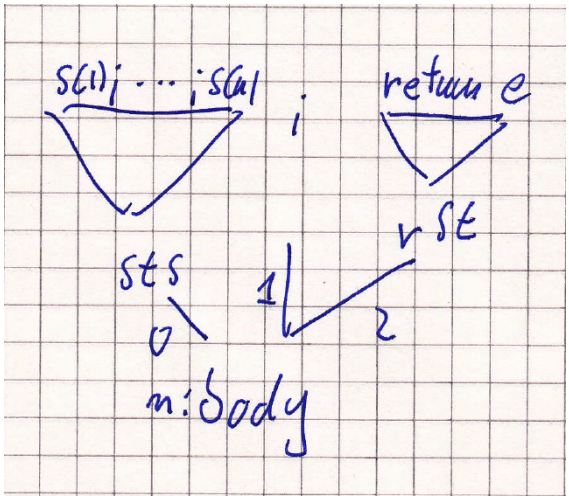


Figure 11.11: Node n generates body $s[1:m]; return \quad e$ of a function. The statement sequence is generated by node $n0$ and the return statement by node $n2$.

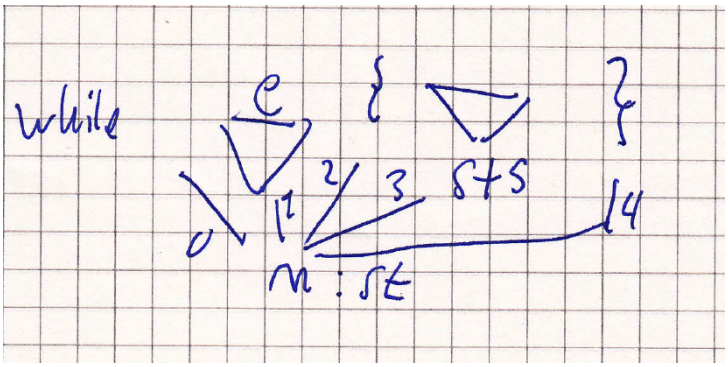


Figure 11.12: Node n generates a while statement. The body is generated by node $n3$.

If then. Let

`hd(c.pr): if e then {if-part}`

We define

$$c'.prn = \begin{cases} fseq(hd(c.prn)4); tail(c.prn) & va(e,c) = 1 \\ tail(c.prn) & va(e,c) = 0 \end{cases}$$

Function call. Let

`hd(c.pr): e = f(e_1,...,e_p)`

We set

$$c'.prn = snodes(f); tail(c.prn)$$

Trivially the invariant

Invariant 8.

$$foralli : c.pr[i] = bw(c.prn(i))\}$$

holds initially and is maintained.
As a second ghost component of $C0$-configurations c we introduce a caller stack

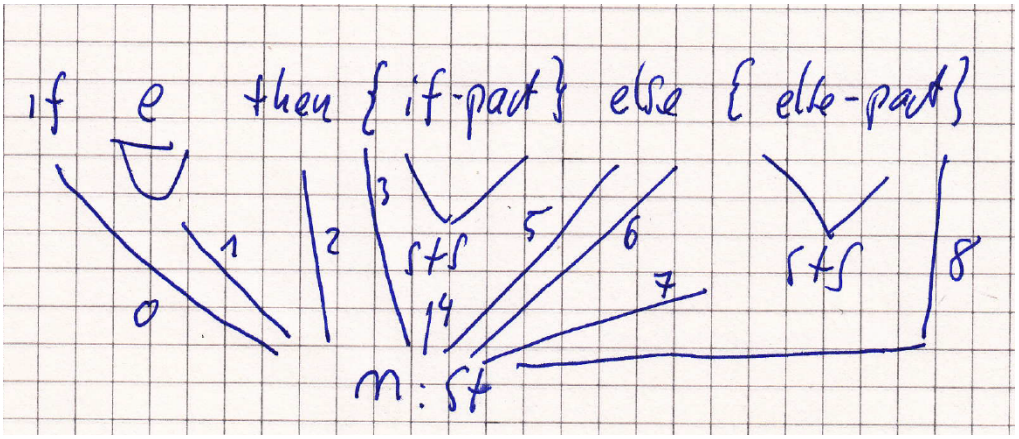


Figure 11.13: Node n generates a conditional statement statement. The if-part is generated by node $n4$ and the else-part is generated by node $n7$.

- $$c.clr : [1 : c.rd] \to T.V$$

In entry $c.clr(i) \in T : V$ of this stack we record the node whose border word generated the function call at recursion depth i. Formally we change this stack only at function calls and returns.

Calls At function calls we have

```
c.pr[1]: e = f(e_1,...,e_p)
```

and

$$c.pr[1] = bw(c.prn[1])$$

We push node $c.prn[1]$ on the caller stack

$$c'.clr(x) = \begin{cases} c.prn[1] & x = c'.rd \\ c.clr[x] & x \leq c.rd \end{cases}$$

Return At the execution of return statements the recursion depth is decreased and the top entry of the caller-stack is dropped

$$c'.clr(x) = c.clr(x) \quad \text{for} \quad x \leq c'.clr$$

We say that c and d are consistent for the caller stack and write $clr - consis(c, d)$ if for all i the second memory word under the base address of $ST(i, c)$ stores the 'return address' where the MIPS program execution should continue after the return of the function. This should be the instruction *behind the end* of the code $code(n)$, that created the stack frame with the function call $bw(n)$. The node n is recorded in $c.clr(i)$. Thus we require

$$d.m_4(ba(ST(i, c) - 328_{32}) = end(code(c.clr(i)) +_{32} 1_{32}$$

Figure 1 illustrates how this works in a configuration c which performs a function call. If we connect alle consistency relations $X - consis(c, d)$ by logical ANDs we obtain the overall consistency relation

$$consis(c, d) = \bigwedge_X X - consis(c, d)$$

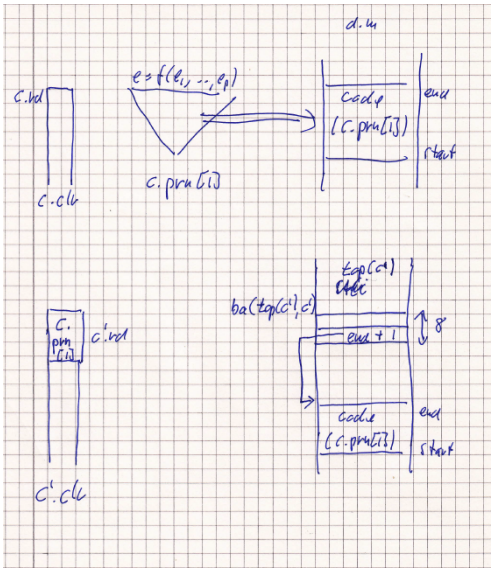


Figure 11.14: Maintaining $clr - consis$ at a function call generated by node $n = c.prn[1]$. The node n is recorded in the new top entry $c'.clr(c'.rd)$ of the caller stack. A new top frame $top(c')$ is created in the MIPS memory. In the second word below the base address we store the return address for the pc after the call. It should 1 byte behind $end(code(n))$.

11.2 Translation of Expressions

11.2.1 Aho-Ullman Algorithm

11.3 Translation of Statements

11.4 Reconstructing Consistent $C0$ configurations from $MIPS$ configurations

Chapter 12

Operating System support in MIPS processors

Bibliography

[KMPar] M. Kovalev, S.M. Müller, and W.J. Paul. *A Pipelined Multi-core MIPS Machine: Hardware Implementation and Correctness Proof*. Springer, 2013, to appear.

[MP98] Silvia M. Müller and Wolfgang J. Paul. On the correctness of hardware scheduling mechanisms for out-of-order execution. *Journal of Circuits, Systems, and Computers*, 8(02):301–314, 1998.

[MP00] S.M. Müller and W.J. Paul. *Computer Architecture, Complexity and Correctness*. Springer, 2000.