Tanenbaum, Austin: *Structured Computer Organization*

Danny Nygård Hansen

31st March 2022

3 • The Digital Logic Level

3.1. Gates and Boolean Algebra

Boolean functions: By definition, a *Boolean function* is a function (in the mathematical sense) $f\colon \{0,1\}^n \to \{0,1\}$, where $n \geq 1$. That is, it takes as inputs n ones and zeros and outputs either 0 or 1.

Binary representation of numbers: Let n be a positive integer. By an *n-bit number* we mean an integer k such that $0 \leq k < 2^n$. In other words, k can be represented as a binary number using n bits. More precisely,

$$k = \sum_{i=0}^{n} a_i 2^i, \tag{3.1}$$

for an appropriate choice of numbers $a_0, \dots, a_{n-1}$ that are either 0 and 1. In this case we write $k = (a_{n-1} \cdots a_0)_2$.

But what if k is not an n-bit number? Say that $k \geq 2^n$. Then we cannot represent k using only n bits, so if k is the result of some calculation using n-bit numbers, e.g. the sum $2^{n-1} + 2^{n-1}$, the most significant bit of k is 'thrown away' unless we explicitly take steps to avoid this. Notice that discarding the most significant bit exactly corresponds to replacing k by its remainder r after division by 2^n: For if k is the $(n+1)$-bit number $(a_n a_{n-1} \cdots a_0)_2$ and $r = (a_{n-1} \cdots a_0)_2$, then

$$k = 2^n + r,$$

which should be obvious but also follows from (3.1).

Arithmetic with positive numbers: Suppose that we have a bunch of numbers and wish to perform a series of algebraic operations on them, e.g. add them together, subtract them from each other, even multiply them (but not divide them by each other). If these operations are performed by a computer, then after each operation we might have discarded the most significant bit (or bits, plural, if we perform a multiplication). Say that we wish to add numbers a, b and c. Doing this 'by hand' would yield the number $a+b+c$, no matter in which order we perform each of the two additions, i.e. whether we compute the number $(a+b)+c$ or $a+(b+c)$. But if we perform these operations on a computer, are we sure that we get the same result regardless of the order in which we perform the additions? After all, we might discard some bits along the way. Are we sure these do not affect the calculation?

The answer should be clear enough: The n least significant bits of $(a+b)+c$ only depend on the n least significant bits of $a+b$ and c. So if we have to throw away a bit when calculating $a+b$, then this has no impact on the n least significant bits of the sum $(a+b)+c$. This means that the n-bit number computed by the computer is exactly the n least significant bits of the sum $a+b+c$ that we computed by hand.

Put in more formal terms, computers perform *modular arithmetic*, to be precise arithmetic modulo 2^n: Write $a \sim b$ if a and b have the same remainder after division by 2^n, or equivalently if 2^n divides $a-b$. If $a_1 \sim a_2$ and $b_1 \sim b_2$, then it is easy to show that $a_1 + b_1 \sim a_2 + b_2$. In other words, whether or not we take the remainder modulo 2^n in our calculations doesn't affect the result, as long as we are only interested in the result modulo 2^n. But as a computer doing calculations on n-bit words, that is exactly what we are.

Negative numbers and two's complement: But what about *negative* numbers? Those are also not n-bit numbers by the above definition. As motivation, consider the difference $a-b$ between two positive numbers. How is a computer supposed to compute this? One idea is to note that $a-b = a+(-b)$. Hence if we can somehow represent $-b$ then we can just compute the sum $a+(-b)$. Of course, this sum might also be negative, so we do in any case need to find some way to represent negative numbers.

Here's the idea: Since computers only care about integers modulo 2^n, it doesn't matter whether we do calulations using the number c or the number $c+2^n$. So if $-2^n \leq c < 0$ and a is nonnegative, then instead of computing the sum $a+c$, let us compute the sum $a+(c+2^n)$. This would then be a sum of nonnegative numbers, and we know how to do that (even if we have to discard a bit after doing the calculation). Therefore, let us use the binary representation of $c+2^n$ as the binary representation of c itself.

So we know how to compute the sum $a+(-b)$, as long as we can compute a representation for the negative number $-b$. That is, given b we need to compute a representation of the number x with the property that $b+x=0$. But by the above discussion, we can instead try to find a binary representation of the number y such that $b+y=2^n$. After all, 0 and 2^n are the same number modulo 2^n. We claim that the following procedure works: Writing $b=(b_{n-1}\cdots b_0)_2$, let $y=(\overline{b}_{n-1}\cdots\overline{b}_0)_2+1$. That is, first we 'flip' all the bits of b (i.e. change ones to zeros and vice versa) and then add 1. Since $b_i+\overline{b}_i=1$, we get

$$b+y=(b_{n-1}\cdots b_0)_2+(\overline{b}_{n-1}\cdots\overline{b}_0)_2+1=(\underbrace{1\cdots 1}_{n\text{ 1's}})_2+1=(2^n-1)+1=2^n.$$

Which is exactly what we wanted. The n-bit binary representation of y is called the *two's complement* of $b_{n-1}\cdots b_0$.

Signed and unsigned integers: If $a_{n-1}\cdots a_0$ is a string of n bits, then we may view this as the n-bit binary number $\sum_{i=0}^{n} a_i 2^i$. In fact, consider a binary string of *any* length, say $a_{m-1}\cdots a_0$ for some (maybe large) integer m. Perhaps this arose as the result of a computation. We can 'by hand' compute the corresponding integer $k=\sum_{i=0}^{m} a_i 2^i$, and by taking its remainder modulo 2^n we obtain what the computer actually computed, namely the string $a_{n-1}\cdots a_0$.

Since the computer only cares about k modulo 2^n, we as humans are free to interpret the string $a_{n-1}\cdots a_0$ as encoding any number on the form $q2^n+k$ for any integer q (positive *or* negative). Notice that we can always choose q such that $q2^n+k$ falls into the interval $[0,2^n-1]$. In this case we say that we interpret the string $a_{n-1}\cdots a_0$ as an *unsigned integer*. Of course, two different n-bit numbers k_1 and k_2 might need different values of q, say q_1 and q_2, such that both $q_1 2^n+k_1$ and $q_2 2^n+k_2$ lie in this interval. On the other hand, we can also choose q such that $q2^n+k$ lies in the interval $[-2^{n-1},2^{n-1}-1]$. In this case we say that we interpret $a_{n-1}\cdots a_0$ as a *signed integer*.

Notice that whether we interpret a binary string as an unsigned or a signed integer has nothing to do with the string per se. It is information *on top of* the information contained in the string. Whether a particular binary string is supposed to represent a signed or unsigned integer is something we either have to keep in mind when looking at the string, or store as extra information about the string, just as we might store that an object in Java has a certain type.

Furthermore, since the computer essentially does calculations modulo 2^n, arithmetic with signed or unsigned integers are performed in the exact same way, i.e. the binary representations should be manipulated in the same way whether we wish to interpret the strings as signed or unsigned integers.

Carry and overflow: Consider two n-bit binary strings $a_{n-1}\cdots a_0$ and $b_{n-1}\cdots b_0$, and consider what happens when we add these together using an ALU. If the most significant bit has a carry out of 1, then we say that the addition results in a *carry*. This is relevant if we interpret the two binary strings as unsigned integers, since then the result of the addition is 'wrong'. If we instead interpret the strings as signed integers there need not be a problem. After all, the sum of a negative and a positive number might be positive, but for this to happen there must be a carry.

However, signed addition might still be problematic, in that it might result in *overflow*. This happens when we add to binary strings that are both positive as signed integers, but whose sum is a binary string that is negative when interpreted as a signed integer. Similarly, if two negative numbers has a positive sum we talk about *underflow*.

Write $\tilde{a} = (a_{n-2} \cdots a_0)_2$ and $\tilde{b} = (b_{n-2} \cdots b_0)_2$ and let $a = a_{n-1}2^{n-1} + \tilde{a}$ and $b = b_{n-1}2^{n-1} + \tilde{b}$. To be clear, all these numbers are regular positive numbers. If adding $a_{n-1} \cdots a_0$ and $b_{n-1} \cdots b_0$ results in overflow, then this means that $a_{n-1} = b_{n-1} = 0$, but that $\tilde{a} + \tilde{b} \geq 2^{n-1}$. Conversely, assume that $a + b < 2^n$ and that $\tilde{a} + \tilde{b} \geq 2^{n-1}$. If either a_{n-1} or b_{n-1} were 1, then we would have

$$2^n > a + b \geq \tilde{a} + \tilde{b} + 2^{n-1} \geq 2^{n-1} + 2^{n-1} = 2^n,$$

which is impossible. So we must have $a_{n-1} = b_{n-1} = 0$. On the other hand, since $\tilde{a} + \tilde{b} \geq 2^{n-1}$ the most significant bit of $a+b$ must be 1. Hence the binary sum of the strings $a_{n-1} \cdots a_0$ and $b_{n-1} \cdots b_0$ is negative as a signed integer, even if both strings are positive as signed integers. Thus we see that overflow happens if and only if $a + b < 2^n$ and $\tilde{a} + \tilde{b} \geq 2^{n-1}$ in the notation above. That is, the sum of a and b is so large that it wraps around and becomes negative, but not so far that it becomes positive again. (Indeed this is impossible, since this would require that $a + b \geq 2^n$, but then either $a_{n-1} \cdots a_0$ or $b_{n-1} \cdots b_0$ must then represent a negative signed integer.)

Next we consider underflow. For this to happen we must have $a_{n-1} = b_{n-1} = 1$, but $\tilde{a} + \tilde{b} < 2^{n-1}$, since otherwise the sum would be negative. Conversely, assume that $a + b \geq 2^n$ and $\tilde{a} + \tilde{b} < 2^{n-1}$. We claim that then $a_{n-1} = b_{n-1} = 1$: At least one of a_{n-1} and b_{n-1} must be 1, since otherwise

$$a + b = \tilde{a} + \tilde{b} < 2^{n-1},$$

contradicting that $a + b \geq 2^n$. If only one of a_{n-1} and b_{n-1} were 1, then we would have

$$a + b = \tilde{a} + \tilde{b} + 2^{n-1} < 2^{n-1} + 2^{n-1} < 2^n,$$

again a contradiction. Hence both $a_{n-1} \cdots a_0$ and $b_{n-1} \cdots b_0$ represent negative signed integers. Notice however that

$$2^n \leq a + b = (2^{n-1} + \tilde{a}) + (2^{n-1} + \tilde{b}) = 2^n + \tilde{a} + \tilde{b} < 2^n + 2^{n-1}.$$

Subtracting 2^n to take the carry into account, the sum is less than 2^{n-1}, so it is a positive signed integer. In total, the addition of $a_{n-1} \cdots a_0$ and $b_{n-1} \cdots b_0$ results in underflow if and only if $a + b \geq 2^n$ and $\tilde{a} + \tilde{b} < 2^{n-1}$.

Notice that overflow and underflow happen just when there is a carry out in precisely one of the two highest bits: Overflow if the carry happens in the second highest, and underflow in the highest. Hence if we want to know whether an addition has resulted in overflow or underflow, we can simply pass the carry out of the corresponding adders through an XOR gate. This does not distinguish between overflow and underflow, but we probably also want to (be able to) check whether there was a carry, which enables us to distinguish between them.

Finally note that some sources do not distinguish between over- and underflow, but instead use the term 'overflow' to refer to both of these phenomena.

$+V_{cc}$: The subscript stands for 'common collector', and the sign indicates that the voltage is positive. Voltage is measured with respect to ground, i.e. the voltage of ground is 0 V.

Current in ≠ current out: It is easy to look at symbols representing gates (e.g. Figure 3-2) and think that the current that enters a gate as an input signal is the same current (i.e. the same electrons) that leaves the gate as the output signal. A moment's thought reveals that this cannot be so: For an input 0 corresponds (in the positive logic, cf. §3.1.4) to no signal at all, so where would the electrons in the output 1 of a NOT gate come from?

Looking e.g. at Figure 3-1(a), we see that each gate is provided with an external power source (here labelled $+V_{cc}$), and it is the current drawn from this source that leaves the gate as an output. (At least this is the case for gates constructed from bipolar transistors.)

3.2. Basic Digital Logic Circuits

Implementing subtraction: Notice that the ALUs of Figures 3.18 and 3.19 do not explicitly implement subtraction. However, since subtraction from B by the number A can be computed by inverting every bit in A, adding this to B, and then adding 1 (i.e. adding the two's complement of A), we may implement the function $(A, B) \mapsto B - A$ by setting both INVA and INC to 1 (see also Figure 4-2).

Also notice that the signal INC is given as the carry in of the first 1-bit ALU in Figure 3.19.

3.3. Memory

Latches: It may be difficult to wrap one's head around how the circuit in Figure 3-21 is supposed to work. If this is so, then the confusion may stem from the same misconception about gates as above: The signal 1 leaving the topmost NOR gate in Figure 3-21(a) is provided by an external power source. In other words, there are no electrons 'going round in circles' or anything of the sort.

Usefulness of D latches: The D latch has an obvious advantage over the SR latch, in that it is not possible to give it the input $S = R = 1$. But we might wonder if the D latch can actually function as proper memory as a result of this restriction: If we want to store the value $Q = 1$ across multiple clock cycles, then we need to supply the input $D = 1$ continually, or at right before the falling edge of each clock cycle. But then we might as well just use the input D instead, so the D latch is not appropriate in this situation.

However, if we only want to store the value of Q for a single cycle, until the next rising edge, then a D latch is appropriate. (TODO: Reference to chapter 4 where we use these things.)

Adding a NOT gate to a flip-flop: In Figure 3-24 we see how to use the delay through a NOT gate to construct a pulse generator. The one shown here produces a pulse on the rising edge. Contemplating Figure 3-24(b) should convince us that placing a NOT in front of the pulse generator will create a pulse on the *falling* edge instead. We might have expected that inserting a NOT gate would simply invert the signal, and this would be the case if gates had no propagation delay. But just as the delay allows the pulse generator to work at all, adding gates in series with it changes its behaviour temporally. Thus we may indeed construct flip-flops that change state on the falling edge as pictured in Figure 3-26(d). (TODO: Also note the placement of the inversion bubble.)

Gates as amplifiers: The 8-bit register in Figure 3-27 has what is seemingly a redundancy: namely a NOT gate on the clock signal before branching out to the flip-flops, followed by a NOT gate immediately before the pulse generator in each flip-flop. These gates each induce a slight delay, but apart from that their effects cancel out, so the behaviour of the circuit shouldn't change if we remove all of these gates.

However, the gates act as amplifiers: If we removed the gates, then the clock signal would have to drive the NOT gates as well as half of each AND gate inside the eight pulse generators. The signal might then be too weak to drive all of these gates. But inserting a NOT gate right before each pulse generator (as well as one in front of the branch point) means that the clock signal only has to drive these eight gates. The external current supplied to these gates would then drive the pulse generators.

But why not just put in a gate that 'does nothing', also called a *(noninverting) buffer* (see below), in front of each pulse generator? Then we wouldn't need the outer NOT gate. It seems that, in practice, a buffer is implemented by placing two NOT gates in series.[1] Hence instead of placing two NOT gates in front of each pulse generator, requiring 16 gates in total, the design in Figure 3-27 only requires 9 gates.

Buffers: We use a slightly different terminology from the book in order to make some distinctions clear. A *(noninverting) buffer* is a logic gate with one data input that implements the identity function. An *inverting buffer* similarly implements inversion, i.e. it is an inverter: a NOT gate.

The buffers in the book furthermore have a control input and are called *tri-state buffers*. Thus a (noninverting) tri-state buffer acts like a normal (noninverting) buffer when the control input is 1, but when the control input is 0 it gives a high-impedance output, which effectively acts like the wire has been cut. This output is neither 0 nor 1, but can be thought of as no output at all, and hence the tri-state buffer has three different outputs or states. An inverting tri-state buffer functions analogously.

Memory organisation: We elaborate on the functioning of the memory in Figure 3-28. Notice that the clock of each flip-flop is being driven both by the control bits (that determine whether or not we should be able to write to the memory) and the address bits. Hence the clock of a given flip-flop is only high if we actively decide to write to it. And between writes it will keep its state, so we do not have to worry about having to maintain an input for its state to be kept.

Notice also that each flip-flop outputs its state continuously, and that the contents of a word in memory is first selected by the address bits and then output by the memory only if appropriate control bits are set. Furthermore, since the output is controlled by tri-state buffers, if the output is not enabled there is no output whatsoever.

4 • The Microarchitecture Level

4.1. An Example Microarchitecture

Binary units:

[1] 'CMOS buffer is formed by cascading two CMOS inverters back to back.' Source.

Decimal			Binary				
Value	SI		Value	IEC		Legacy	
1000	k	kilo	1024	Ki	kibi	K	kilo
1000^2	M	mega	1024^2	Mi	mebi	M	mega
1000^3	G	giga	1024^3	Gi	gibi	G	giga
1000^4	T	tera	1024^4	Ti	tebi	T	tera
1000^5	P	peta	1024^5	Pi	pebi	–	–
1000^6	E	exa	1024^6	Ei	exbi	–	–
1000^7	Z	zetta	1024^7	Zi	zebi	–	–
1000^8	Y	yotta	1024^8	Yi	yobi	–	–

Words and addressing: The *word size* of an architecture is the number of bits that the CPU can process at one time. A *word* is a 'word size'-bit number, and these are handled as a unit by the instruction set or the hardware. The registers of the processor are usually word-sized, and words are often the largest data that can be transferred to and from memory in a single instruction. The Mic-1 is a 32-bit architecture since its registers contain 32 bits, and it can read and write 32-bit strings to and from memory.

When we read to and write from memory, we must distinguish between the address of a piece of memory and its contents. An *address* is just a number, of course represented as a binary string in practice, which uniquely identifies some portion of memory. The number of addresses is usually bounded by the word size, since we want the ability to store a single address in a register. That is, since there are 2^n different n-bit numbers, if the word size is n then there are 2^n possible addresses.

In contrast, each address must identify some portion of memory with some size. If each address identifies a byte, then we say that the computer is *byte-addressable*. If each address identifies a word, then we say that the computer is *word-addressable*. Most modern computers are byte-addressable. Since words are [TODO: always?] larger than bytes, this puts a limit on the size of addressable memory. For instance, if we require that a machine with word size of 32 be byte-addressable, then the 2^{32} different addresses can point to 2^{32} individual bytes, in total

$$2^{32}\,\text{bytes} = 4 \cdot 2^{30}\,\text{bytes} = 4\,\text{GB},$$

or more properly 4 GiB. This helps explain why 32-bit computers only support 4 GB of memory. On the other hand, being able to address smaller units of memory makes it easier to work with small items of data. For instance, we often want to store distinct units (like numbers) at different addresses, but if the numbers are much smaller than the *minimal addressable unit* (MAU), then there might be a lot of wasted space.

Note however also that the MAU is a property of a specific memory *abstraction*. For example, the Mic-1 microarchitecture has a 32-bit, word-addressable port, controlled by the MAR and MDR registers. These are not able to address units smaller than (32-bit) words, even though they access the same underlying memory.

Registers in Mic-1 data path:

- MAR: Memory Address Register
- MDR: Memory Data Register
- PC: Program Counter
- MBR: Memory Buffer Register
- SP: Stack Pointer
- LV: Local Variable
- CPP: Constant Pool Pointer
- TOS: Top of Stack
- OPC: Opcode Counter
- H: Holding register

The flags N *and* Z: Each clock cycle, the ALU performs some operation (whether we ask it to or not). The sign bit (i.e. the most significant bit) of the result is then set as the status flag N. In other words, N is set to 1 just if the result is negative. It is also checked whether the result is zero, and if so we set the status flag Z to 1. These status flags are then stored in registers of the same names, as described below.

Compare the FLAGS register of the x86 architecture.

Data path cycles: We elaborate on what happens during a single data path cycle. Before the cycle begins, we assume that the MPC register has been loaded with the correct address, and that the input to and output from the control store is stable (we will see later how this happens). The cycle then begins on the falling edge of the clock, and we can think of it as composed of four subcycles:

(1) The contents of the control store at the address stored in MPC are read into the MIR register.

As we know, the control store need not actually contain a microprogram in software, but may implement (some of) its functional in hardware, i.e. using logic gates. But assume that it is a memory similar to the kind discussed in §3.3.4, in particular in Figure 3-28. In the notation in the figure, the contents of MPC are directed into the address inputs $A_0, A_1, \ldots, A_8$. Since we do not wish to ever write to the control store, we can output the contents of the addressed word continuously and do not have to worry about the control bits to the memory. The register MIR can then be chosen to be triggered on the falling edge of the clock. Note that the falling edge is long enough that it is possible to load the register during this time, cf. §3.3.2.

We wait a time Δw until MIR has been loaded. Since signals have to pass through a series of gates, there is a certain propagation delay. After this interval has passed, MIR will output its contents.

(2) The signal propagates into the data path, selecting which register to put onto the B bus, and we also already select which register(s) to write to at the end of the cycle. Since we do not write to any registers yet, we do not have to time this with the clock. This takes a further time Δx.

(3) At the same time, control signals have been propagating to the ALU and the shifter. These now also receive their correct data inputs from the B bus and the H register. Since both the ALU and the shifter are combinatorial circuits, we do not have to worry about their inner state. They have been computing some function throughout the above execution, but it is only now that they have both the correct control and data inputs. Now they compute the correct function of the correct data, and it takes a time Δy for the outputs of the ALU and the shifter to stabilise.

(4) The output of the shifter propagates along the C bus back to the registers. This takes a time Δz.

After the time $\Delta w + \Delta x + \Delta y + \Delta z$ has passed, the input to the registers from the C bus is stable. When the MIR register was loaded, control bits already started propagating to each register in the data path, so they now both have the correct control bits set and the correct data inputs. On the rising edge of the clock they then store their inputs, and at the same time the inputs to the N and Z are stored.

Since the ... TODO TODO

The fourth cycle ends a little after the rising edge.

After the rising edge of the clock, we need to determine the next instruction to execute, i.e. find the address of this instruction so that we can load it from the control store. To do this we need the contents of the registers MIR, N, Z ... TODO TODO

Branching: Each microinstruction must contain the address of the next microinstruction to execute, so unless we are somehow able to alter this address during the execution of the microinstruction in question, every microprogram will be deterministic[2]. Altering this address is called *branching*, and several elements determine whether to branch and, if so, where to branch to.

During the execution of the ALU, it checks whether the input from the B bus is negative, zero, or positive. If it is negative, the status flag (which is an output) N is set to 1, and Z is set to 1 if the input is zero. This happens in every data path cycle.

We can then use the values of N and Z to decide whether to branch. This is done using the bits JAMN and JAMZ in the microinstruction. The block in Figure 4-6 labeled 'High bit' computes the Boolean

$$(\mathsf{JAMN} \wedge \mathsf{N}) \vee (\mathsf{JAMZ} \wedge \mathsf{Z}) \vee \mathsf{NEXT\_ADDRESS}[8].$$

That is, the bits JAMN and JAMZ decide whether to branch on N and Z, and the way we branch is to set the most significant bit of NEXT_ADDRESS to 1. Notice that the two potential address we branch two must differ only in their most significant bit (i.e. as decimal numbers they must differ by $2^8 = 256 = 0x100$). We can choose to set either one of JAMN and JAMZ, none, or both. When we implement IJVM we will not have use for setting both at the same time, but the microarchitecture does not prohibit this.

Another way to branch is to set the bit JMPC. While JAMN and JAMZ only let us branch to addresses stored in the NEXT_ADDRESS bits of microinstructions already in the micropogam (up to flipping a bit), JMPC allows us to use any address we read from the main memory: The computation is done in the block labeled 'O' in Figure 4-6. If JMPC is set to 1, then we do an OR of MBR with NEXT_ADDRESS and send the result to MPC. If instead JMPC is set to 0, then we just pass NEXT_ADDRESS directly through to MPC. Since MBR can hold any 8-bit number, this register alone lets us access $2^8 = 256$ addresses in the control store. If we set JMPC to 1, then we will usually set NEXT_ADDRESS to either 0x000 or 0x100. In this way, by OR'ing together JMPC and NEXT_ADDRESS we can access all 512 addresses in the control store. [TODO: Will we ever have to set both JAMN or JAMZ, as well as JMPC?]

We will use this in the following way [TODO: Reference to when we see this in practice later.]: Each ISA instruction will be implemented using multiple microinstructions. An ISA instruction (or more precisely its opcode) will be loaded into MBR, and this instruction is just a pointer into the control store. If JMPC is set to 1, when jumping to the next microinstruction the microprogram will thus go to the corresponding address in the control store. The microinstruction located here will have a NEXT_ADDRESS, and so will the next microinstruction, and eventually the ISA instruction we loaded initially

[2] In the sense that the execution of the microprogram neither depends on the main memory nor on the output of the ALU.

will be finished. In the meantime we have read the next ISA instruction from memory, and we are ready to execute this by setting JMPC to 1 again.

Note that Figure 4-6 is slightly misleading: The output from the 'O' block is 9 bits long, but the figure shows it only taking up the 8 least significant bits of MPC.

4.2. An Example ISA: IJVM

IJVM syntax:

- Assembler directives: Instructions beginning with a full stop . . Strictly not part of IJVM itself, but directives that alter the behaviour of the assembler/compiler. Compare preprocessor directives in C, e.g. `#include` and `#define`.
- Methods: Declared with the directive `.method <name>`. Every IJVM program must have a `main` method.
- Method arguments: Declared with `.args <number>`. Placed on the line below the `.method` directive. The number of arguments includes the 'dummy' argument `OBJ_REF`, which is overwritten with the link pointer after the method has been called.
- Local variables: Declared with `.locals <number>`, also placed after `.method` but before the body of the method.
- Definitions: Declared with `.define <name> = <value>`. Essentially replaces all occurrences of `<name>` with `<value>` throughout the program. This does not allow any behaviour not already allowed.

Experimenting with memory: Let us first consider the simplest possible IJVM program:

```
.method main
.args 1
    ireturn
```

Since the program takes no arguments, we may omit the directive `.args 1` if we wish. Compiling this to bytecode with `ijvm-asm` and excuting yields the output

```
                                 stack = 0, 1, 4
ireturn              [ac]        stack = 0
return value: 0
```

How do we understand the 4 at the bottom of the stack? Let us write out the contents of the bytecode file:

```
main index: 0
```

```
method area: 5 bytes
00 01 00 00 ac
constant pool: 1 words
00000000
```

We conjecture that (some of) this data is stored at memory addresses smaller than 4. First notice that writing a program whose method area is exactly 8 bytes does not increase the number at the bottom of the stack. Take for instance the program

```
.method main
.args 1
    nop
    nop
    nop
    ireturn
```

This yields a similar output to the first program, namely

```
                               stack = 0, 1, 4
nop                 [00]       stack = 0, 1, 4
nop                 [00]       stack = 0, 1, 4
nop                 [00]       stack = 0, 1, 4
ireturn             [ac]       stack = 0
return value: 0
```

However, adding a single extra nop instruction does result in an increase:

```
                               stack = 0, 1, 5
nop                 [00]       stack = 0, 1, 5
nop                 [00]       stack = 0, 1, 5
nop                 [00]       stack = 0, 1, 5
nop                 [00]       stack = 0, 1, 5
ireturn             [ac]       stack = 0
return value: 0
```

Thus indicating that the stack is located after the method area in memory.

We may perform a similar experiment on the constant pool. Consider the program

```
.method main
.args 1
    ldc_w 42
    ireturn
```

This yields the following output:

```
                               stack = 0, 1, 5
ldc_w 1             [13 00 01]  stack = 42, 0, 1, 5
ireturn             [ac]       stack = 42
return value: 42
```

Looking at the bytecode we see that, since `ldc_w 1` has a three byte opcode [TODO: Terminology, do arguments count as part of the opcode? Probably not.], the method area is still eight bytes long, but the constant pool has grown:

```
main index: 0
method area: 8 bytes
00 01 00 00 13 00 01 ac
constant pool: 2 words
00000000
0000002a
```

Further experiments with adding one-byte instructions such as nop show that, as expected, the number at the bottom of the stack increases whenever the number of bytes in the bytecode increases to one more than a multiple of 8.

Can we understand why we get the exact value we do? The number is supposed to be the address of the previous PC (whatever 'previous' means when talking about `main`), and this is presumably the number 1 on the stack. For instance, in the first program above we needed two words for the method area, and one word each for the main index and the single word in the constant pool. In total four words, and we also need space to store the link pointer itself. Thus we might have expected the link pointer to have the value 5, but we seem to be off by one. I am not sure why that is, perhaps the main index is not stored here or something?

Next, from Figure 4-12 we see that the bottom-most element of the stack is supposed to contain the *link pointer*, i.e. a reference to the previous program counter (again, whatever 'previous' means) Above the program counter we should find the previous LV, which is a pointer to the bottom of the local variable frame of the previous method. And indeed, increasing the number of arguments or local variables also increase the value of the link counter.

Calling methods: Let us now see what happens to the stack when we call a method. Consider the following program:

```
.method main
.args 1
    bipush 42
    invokevirtual test
    ireturn

.method test
.args 1
    ireturn
```

Notice that we push the dummy value 42 to the stack as an object reference before calling `test`. This produces the following output:

```
                                    stack = 0, 1, 7
bipush 42           [10 2a]      stack = 42, 0, 1, 7
invokevirtual 1     [b6 00 01]   stack = 6, 9, 10, 0, 1, 7
ireturn             [ac]         stack = 6, 0, 1, 7
ireturn             [ac]         stack = 6
return value: 6
```

First of all, notice that the link pointer contains the address 8, consistent with the fact that both the constant pool and method area are larger, since they have to contain `test`.

Address	Stack	Description
8	0	Previous LV.
7	1	Previous PC.
6	7	Link pointer.

Notice how we have computed these addresses: Since there are no method parameters or local variables, the link pointer references the word directly above itself, so the link pointer must have address one less than its value.

Address	Stack	Description
11	6	Caller's LV.
10	9	Caller's PC.
9	10	Link pointer.
8	0	Previous LV.
7	1	Previous PC.
6	7	Link pointer.

We see that the caller's LV, stored in address 11, does in fact contain the address of the previous link pointer, which is where the caller's LV is supposed to point to. Can we understand the caller's PC as well? This is supposed to contain a byte address, so it does not refer to the *word* address 9 (clearly, since this contains the link pointer as we see in the table). So what does it refer to? Consider the bytecode:

```
main index: 0
method area: 15 bytes
00 01 00 00 10 2a b6 00 01 ac 00 01 00 00 ac
constant pool: 2 words
00000000
0000000a
```

We see that addresses 6-8 contain the opcode [TODO: Opcode plus arguments?] for `invokevirtual`. It seems that the caller's program counter has been incremented just before calling `test` and now points to the opcode for the following `ireturn` instruction.

Let us use this knowledge to break an IJVM program. Consider the following program:

```
.method main
.args 1
    bipush 42           // Dummy value for OBJ_REF
    invokevirtual test
    bipush 43           // Is never executed
    ireturn

.method test
.args 1
    pop                 // Pop caller's LV ...
    bipush 2
    iadd                // Skip the instruction for bipush 43
    bipush 8            // ... and push caller's LV
    ireturn
```

Running the program yields the stack trace

```
                                stack = 0, 1, 9
bipush 42           [10 2a]     stack = 42, 0, 1, 9
invokevirtual 1     [b6 00 01]  stack = 8, 9, 12, 0, 1, 9
pop                 [57]        stack = 9, 12, 0, 1, 9
bipush 2            [10 02]     stack = 2, 9, 12, 0, 1, 9
iadd                [60]        stack = 11, 12, 0, 1, 9
bipush 8            [10 08]     stack = 8, 11, 12, 0, 1, 9
ireturn             [ac]        stack = 8, 0, 1, 9
ireturn             [ac]        stack = 8
return value: 8
```

Let us go through the execution of the program: We first push the dummy value 42 and call `test`. Now the caller's LV is at the top of the stack, but we would like to access the caller's PC below it. So we pop and manually take note that the topmost element of the stack was 8. Now the caller's PC is the topmost element. We push 2 and add, essentially incrementing the callers's PC twice, which has the effect of skipping the instruction `bipush 43` in the body of `main`. Now we manually restore the caller's LV by pushing 8, and then we return.

And indeed, we see that the instruction `bipush 43` is never executed.

Registers in IJVM:

4.3. An Example Implementation

Labels in MAL:

Branching in MAL: As described above [TODO: Reference?] we can branch depending on the output of the ALU. Say that we perform some calculation and save the result, e.g. `OPC = TOS + H`. We are then able to branch depending on whether TOS + H is negative, using the N register, or whether it is zero, using the Z register. Say we wish to branch to the label L1 if N is set, and branch to L2 if not. The syntax for this is

```
OPC = TOS + H; if (N) goto L1; else goto L2
```

Note that the statement `if (N) goto L1; else goto L2` is a single statement and cannot be broken up: Whenever we branch in this way, we must provide both an ‘if’ and an ‘else’ clause.

We might expect to be able to break this statement into two, by splitting it into two lines:

```
OPC = TOS + H
if (N) goto L1; else goto L2
```

However, this will *not* have the same behaviour as keeping the statements on the same line. Each line of MAL code corresponds to one data cycle. The one-line version of the branch specifies that a certain ALU operation be performed, namely an addition, and the result of this operation be used to decide whether or not to branch. This all happens in the same data cycle. In contrast, the two-line version takes two cycles to complete: In the first cycle an addition is performed. In the second cycle we haven’t specified which ALU operation to perform, so the assembler presumably chooses either a default operation or uses garbage data since the result isn’t stored anywhere and thus doesn’t matter. Whichever it is, it is *not* the result of the operation `OPC = TOS + H` that is used to decide whether or not to branch.

As a side-note, there might be situations where we want to use the result of the same ALU operation (that is, the same type of operation with the same input values) to decide whether to branch multiple times in a row. For instance, we might want to branch one if the result is negative vs. nonnegative, and then branch another way if the same result is zero vs. nonzero. Thus it might be tempting to reuse the same computation in both branches. But not only is that not possible, it also wouldn’t make the program any more efficient: For given the microarchitecture, we are only able to choose between two addresses when branching, so if we wish to jump between more than two addresses, then we need multiple instructions. But in every data cycle we perform an ALU operation anyway, so we might as well perform the same operation multiple times in a row. In other words, the control unit has to wait for the ALU anyway.

A • Notes on C

Memory allocation: The memory used by a C program is often said to be divided into two: the *stack*, which is used for local variables and function arguments, and the *heap*[3], on which we can allocate memory dynamically, perhaps if we wish to store an array larger than can fit on a stack.[4] However, *nothing* in the C standard requires there to even be a stack or a heap anywhere. In fact, local variables are often placed in registers by optimising compilers instead of being placed on the stack.

In practice, the stack is usually used more or less as in the IJVM memory model: When a function is called, a block of memory is reserved on top of the stack for local variables and other information, and this block is popped when the function returns. In contrast, the heap is indeed usually used for dynamic allocation.

The C type system: In C, *variables* have types but *values* do not. When a variable is declared, for instance by writing

```
int n;
```

then the compiler understands that the name `n` refers to a value of type `int`. The compiler can use this information in the usual ways to ensure that the correct types are given as arguments to functions, to implement random access into arrays, etc.

At some point the variable will likely be associated with some value located either in memory or in a register. But the bits at this location do not contain information about the *type* of information stored there, nor is this information stored anywhere else. In particular there is no way to check the type of a variable at runtime, types are only available at compile-time.

[TODO: Strictly speaking there is, to my understanding, nothing in the standard that prohibits variables from having types at runtime. But then again, the standard of course includes no function to check the type of a variable, so one would need to provide this on top of the functions given in the standard.]

Note that a pointer is not just a pointer, it is a pointer *to something*.

Function prototypes: Depending on the compiler, you might be able to call a function before it is declared, but the compiler is required by the C11 [TODO: Or is it C99?] standard to complain in some way that this is happening. So why not just declare (and simultaneously define) every function before it is used?

[3] Note that there is no obvious connection between *the* heap, the place in memory, and *a* heap, the data structure. This is confusing since *the* stack is certainly *a* stack.

[4] Running `ulimit -a` in a Linux terminal gets user limits, such as the maximum size of a program stack.

Sometimes this is not possible, for instance if two functions are mutually recursive.

The solution is to use *function prototypes*, which is a declaration of the function's return and argument types. If this is placed at the top of the file, the functions can be used when later *defining* each function. Function prototypes are recommended whether or not they are strictly necessary.

Argument promotion: Arguments to a function with a prototype are implicitly converted to the types of the corresponding parameters. If the function does not have a prototype, the *default argument promotions* are used, in which expressions of type `char` or `short int` are automatically promoted to `int`.

Variadic functions: Some functions take a variable number of arguments, for instance `printf`. These are usually called *variadic* functions, though this terminology is not found in the C11 standard. According to *The GNU C Library Reference Manual*, a variadic function must be declared with a prototype that says that it is variadic, which is done with an ellipsis '`...`' appearing last in the parameter list, following at least one named parameter. Arguments corresponding to the ellipsis seem to be referred to as *variadic arguments*, but this also does not seem to be an official term.

Arguments corresponding to the named parameters of a variadic function (which has a prototype) are implicitly converted as with non-variadic functions, and the variadic arguments undergo default argument promotion (C11, 6.5.2.2, p. 7).

The function `sizeof`: The function `sizeof` has the return type `size_t`, which is specifically used to represent the size of objects. It is guaranteed to be large enough to contain the size of the largest object the system can handle.

If we wish to print a value of type `size_t`, we therefore use the length modifier `z` along with the conversion specifier `u`, since the value is unsigned. This only works if the compiler supports C99 (which it almost certainly does), but if it doesn't then `lu` also works. The compiler won't complain, but it is probably good style to explicitly cast to `unsigned long` before printing. (Actually, on some systems it might be necessary to use `llu`, if an `unsigned long` is not big enough to represent the largest object for the given environment.)

Printing pointers: Note that the conversion specifier `p` expects a value of type `void*` (see C11, 7.21.6.1, p. 8). Strictly speaking, giving as argument to `printf` another type of pointer, e.g. `int*`, results in undefined behaviour, and hence is implementation-dependent. A warning is produced if compiled with the flag `-Wpedantic`. So, again strictly speaking, casting to `void*` is required but it is not usually necessary in practice. [TODO: Has something to do with POSIX?] (Note, however, that casting to and from `void*` more generally is done implicitly.)

The reason this is necessary is that the C standard does not require that pointer types have the same size or representation.[5] In practice a `void*` and an `int*` have the same size (for instance 64 bits on a 64-bit machine) and are represented in the same way, but the standard does not require this, and there is no guarantee that this will not change in the future. And even if they were practically identical, there is no guarantee that the compiler will treat them the same.

Double pointers: Since a pointer has a memory address, one can also consider pointers to pointers, denoted by two asterisks `**`. But there is nothing special about these objects, they are just pointers to a type that happens to itself be a pointer type.

Incomplete types: [TODO]

Structs: We distinguish between *defining* and *declaring* structs. The minimal syntax for defining a struct is

```
struct {};
```

This defines a struct with no name, so it cannot be used subsequently. There is really no reason to do this. We may also give the struct a name, e.g.

```
struct s {};
```

which defines a struct with the name `s`. To declare a variable of this type we write

```
struct s t;
```

We may also use the struct definition immediately when declaring a variable:

```
struct {} t;
```

In this case it can make sense not to give the struct a name, but this precludes us from declaring other variables of the same type. We may instead give the struct a name *and* declare the variable `t` with the corresponding type in the same declaration:

```
struct s {} t;
```

The braces in all of the above cases may of course contain variable declarations and so on as usual. Members of incomplete types are not allowed.

After the struct has been named it has incomplete type: it is only when the entire struct definition is complete that it has complete type. Since the variable declarations inside a struct definition do not allow incomplete types, the following is *not* allowed:

```
struct s { struct s t; };
```

[5] What is true is that pointers to *compatible* types have the same representation, see C11, 6.2.5 p. 28.

However, pointers to incomplete types *are* allowed, so the following is legal:

```
struct s { struct s *p; };
```

Hence it is possible to create (effectively) recursive structs.

Instead of defining a struct we may also *declare* it. A declaration on the form

```
struct s;
```

is called a *forward declaration*. This declares `s` as a new struct name in the scope (hiding any previously declared meaning for the name), but it does not define it. Until it has been defined, using the syntax above, the struct name has incomplete type. This syntax is useful in defining mutually recursive structs, e.g.

```
struct r;
struct s { struct r *p; };
struct r { struct s *q; };
```

Notice that the definition of the struct `s` is legal since *pointers* to incomplete types are legal in struct definitions. Compare also function prototypes mentioned above.