ECE M116C

Warren Kim

Contents

Part I

First 5 Weeks

Chapter 1

Preface

1.1 Abstraction

Definition: Abstraction

Abstraction is the concept of providing a (relatively) simple interface to higher level programs, hiding unnecessary complexity.

We define a computer as a black box with multiple layers of *abstraction*. When focusing on a particular layer, we abstract away the irrelevant layers. There are three main abstraction layers:

(i) Application Layer: Here, we translate from algorithms to code. We usually write these in high level languages (e.g. C/++, Java, etc.).

(ii) Systems Layer: Here, we have the compiler that translate HLL[1] code to machine code, and the operating system, which deals with everything you learned in CS111.

(iii) Hardware Layer: The hardware layer is the physical hardware (who would've thought) that makes all of this possible (e.g. CPU, RAM, etc.)!

Similar to computers, program code also has three layers of abstraction!

(i) High-Level Language: This layer hosts all of our favorite languages (e.g. C/++, JS, etc.). This level of abstraction provides good productivity and portability[2].

(ii) Assembly Language: A textual representation of hardware instructions. Assembly is architecture dependent!

(iii) Hardware Representation: Here we have the actual binary (1's and 0's) that encode instructions and data.

1.2 Instruction Set Architecture

Definition: Instruction Set Architecture

The **Instruction Set Architecture (ISA)** is the set of instructions supported by a computer. There are multiple (all incompatible) ISA's and they usually come in families. ISA's usually come with privilaged and standard instruction sets.

The ISA is an *interface* between hardware and software, and as such, allows them to develop and evolve *independently*.

The software sees a *functional* description of the hardware: *(i)* Storage locations (e.g. memory) and *(ii)* Operations (e.g. `add`).

The hardware sees a list of instructions and their *order*.

[1]HLL: **H**igh **L**evel **L**anguage.

[2]Most languages are hardware/architecture agnostic.

1.3 Efficiency

The main objective when architecting a computer is to make it *efficient*. Here, we define "efficient" to be:

(i) Performance[3]: Fast.

(ii) Power Consumption: Low.

(iii) Cost: Low.

(iv) Reliable and Secure.

1.4 History

Look at the slides for the full history. TLDR: we've come a long way. The main takeaway of this section is that we've been able to make these improvements for two major reasons: *new technologies* and *innovative techniques*.

1.5 Laws

1.6 Moore's Law

Definition: Moore's Law

Moore's Law states: "The number of transistors in an IC[a] doubles every two years."

[a]IC: Integrated Circuit.

This Moore guy is pretty smart since we've been roughly on track with his prediction (up until about 2005).

Definition: Dennard's Scaling Law

Dennard's Scaling Law states: Moore's law $\implies$ each transistor's area is reduced by 50% (or every dimension by 30%).

Naturally, it follows that:

(i) Voltage is reduced by 30% to keep the electric field constant (remember $V = EL$? Me neither.).

(ii) L is reduced $\implies$ delays are reduced by 30% ($x = Vt$).

(iii) Frequency is increased by 40% ($\textit{frequency} = \frac{1}{\textit{time}}$).

(iv) Capacitance is reduced by 30% ($C = \frac{kA}{L}$).

(v) Since $P = CV^2f$ (apparently), power consumption per transistor is reduced by 50%. As such, the power consumption of the entire chip stays the same.

Definition: Amdahl's Law

Amdahl's Law states: "The performance improvement (*speed up*) is limited by the part you cannot improve (*sequential part*)." That is,

$$\textit{speed up} = \frac{1}{(1-p) + \frac{p}{s}}$$

where p is the part that can be improved and s is the factor of improvement.

[3]Performance is *usually* the most important metric.

1.6.1 The Power Wall

Up until 2005, we've been able to make transistors smaller (ergo faster) while keeping power consumption the same. Unfortunately, since 2005, due to *tiny* transistor sizes, the static power leakage has become so dominant that we couldn't keep the power consumption the same.

1.6.2 Multi-Core Era

Guess what's better than one CPU core? Multiple! Unfortunately, Amdahl is a party pooper and his law suggests we're hitting peak performance.

Chapter 2

Instruction Set Architecture

Definition: Instruction Set Architecture

The **Instruction Set Architecture (ISA)** is the set of instructions supported by a computer. There are multiple (all incompatible) ISA's and they usually come in families. ISA's usually come with privilaged and standard instruction sets.

The ISA is the contract between software and hardware, and is typically defined by giving all the programmer-visible state (registers and memory) as well as the semantics of the instructions that operate on that state. [1]

As described in the definition, many implementations of a given ISA are possible. Here are a few:

(i) AMD, Intel, VIA processors run the AMD64 ISA.

(ii) (Most) cellphones use the ARM ISA with varying implementations from companies including (but not limited to): Apple, Qualcomm, Samsung, Huawei

Corollary: Design Methodology

ISA's are typically designed with particular micro-architectural style(s) in mind. Here are some examples:

(i) Accumulators $\rightarrow$ hardwired, unpipelined.

(ii) CISC $\rightarrow$ microcoded.

(iii) RISC[a] $\rightarrow$ hardwired, pipelined.

(iv) VLIW $\rightarrow$ fixed-latency in-order parallel pipelines.

(v) JVM $\rightarrow$ software interpretation.

However, they can be implemented with any micro-architectural style. Here are some examples:

(i) Intel Ivy Bridge: Hardwired pipelined CISC (x86) machine (with some microcode support).

(ii) Spike: Software-interpreted RISC-V machine.

(iii) ARM Jazelle: A hardware JVM processor.

[a]In this class, we'll focus on this one!

[1]Note: The IBM 360 was the first line of machines to separate ISA from implementation; i.e. *microarchitecture*.

2.1 RISC-V

RISC-V is an open-source[2] "RISC"-based[3] ISA (royalty-free) that was developed in the 2010's at Berkeley. In this class, we'll focus on the 32-bit "base" mode; i.e. "RV32I".

It is an alternative to CISC[4], with the main differences highlighted below:

RISC

(i) Fixed instruction size.

(ii) Simple (one-by-one) operation.

(iii) Less Complex.

CISC

(i) Variable instruction size.

(ii) Packed operation.

(iii) Complex (it's in the name).

Corollary: Widely Used

With the exception of x86 (Intel's ISA) and a few others, all ISA's are based off of RISC, including (but not limited to) MIPS, ARM, PowerPC, RISC-V.

2.2 Running Instructions

2.2.1 Stored Program Computer (Von Neumann)

Definition: Von Neumann Architecture

Computer hardware is a machine that reads instructions one-by-one and executes them *sequentially*. It continues this until the program terminates or finishes.

Memory Integration

Memory holds *both* the program (set of instructions) as well as the data it uses/manipulates in a linear memory array. Because of this, they can be modified during program execution, allowing for flexibility/more complex software design.

Sequential Instruction Processing

Definition: Program Counter

A **Program Counter (PC)** is a register that contains the address of the current instruction.

We do the following to execute a set of instructions:

(i) The PC *identifies* the current instruction.

(ii) We *fetch* the next instruction from memory.

(iii) We *update* the state (e.g. PC and memory) as a *function* of the current state according to the instruction.

(iv) *Repeat* until the program terminates.

[2]It is currently mostly maintained by the open-source community.
[3]RISC: **R**educed **I**nstruction **S**et **C**omputers.
[4]CISC: **C**omplex **I**nstruction **S**et **C**omputers

2.3 Building an ISA: Operands

Instructions in 32-bit RISC-V take the form:

```
COMMAND OPERANDS
```

where each instruction is fixed-size and 32-bit. Possible types of `OPERANDS` are:

(i) Registers

(ii) Immediate

(iii) Memory

In RISC-V, we will have *at most* 2 operands.

2.3.1 Registers

Definition: Register

A **register** is a small storage unit *inside* the processor to quickly access data, addresses, and instructions. It is typically smaller (and as such, faster) than memory, and can be seen by software[a]. There are typically between 16 and 64 registers in modern ISA's.

[a] I guess we'll clarify this later. (Source: Lec. 2 Slide 24)

Definition: Register Width

The **width** of a register refers to the number of bits it can hold.

A larger register width $\implies$ more bits can be processed simultaneously, allowing for faster data processing and transfer rates. This comes at the cost of power consumption: wider registers $\implies$ more electronic circuits are activated at once $\implies$ higher power consumption[5].

RISC-V Registers

RISC-V supports 32 registers and 2 sizes:

(i) A *word* has a width of 32 bits.

(ii) A *double-word* has a width of 64 bits.

Aside: Floating Point Registers

Note that all 32 registers store values in *integers*. Higher-end processors may include a *separate* set of registers for floating point operations.

Each register is denoted by `xi`, where `i` is an integer between 0 and 31 inclusive.

Note: $x0$ is *hardwired* to 0; i.e. it will *always* contain the value 0[6].

Note: Registers are stored in a data structure called the *register file* (which will be discussed "later" [Source: Lec. 2 Slide 28]).

[5] Low-end processors use smaller registers. High-performance processors use wider and more registers.
[6] This is useful for various operations when we need 0; e.g. resetting other registers.

2.3.2 Immediates

Definition: Immediate

An **immediate** is a *signed* constant number used in an instruction.

Example: `addi`

Consider the following:

```
addi x2, x1, 5
```

Here, 5 is the immediate that is being used in the `addi` instruction. We can infer that this command adds two numbers, the value in `x1` and 5, and stores the result in another register `x2`.

Corollary: IMPORTANT: Immediate Sizes

While registers are fixed at 32 bits, immediates ***need not be*** 32 bits in size; i.e. they are *variable* size. Why? The explanation given in class was that there is simply no room to store a 32-bit immediate.

Sign

Since many operations in RISC-V are signed, proper sign-extension is needed when necessary. There are two cases:

Case (i) LSB: Here, padding zeroes is efficient.

Case (ii) MSB: Here, we need proper sign extension (remember 2's complement? me neither).

2.3.3 Memory

Memory contains data that can be accessed by the processor.

Definition: Load and Store

Loading is the operation of *reading* from memory.
Storing is the operation of *writing* to memory.

There are a couple of variations for loading and storing:

(i) `LW`: Load Word

(ii) `LB`: Load Byte (is implemented by doing LW and only reading the *highest* byte of data).

(iii) `LH`: Load Half-Word (is implemented by doing `LW` and reading the *lower* half of the data.

(iv) `LBU`: Load Byte Unsigned (padding with 0's)

(v) `LHU`: Load Half-Word Unsigned (padding with 0's)

(i) `SW`: Store Word

(ii) `SB`: Store Byte

(iii) `SH`: Store Half-Word

Format and Addressing

Definition: Base + Offset

Typically, we address memory using the following formula:

$$addr = x_i + offset$$

where x_i is the base address.

We do this because immediates cannot represent larger addresses.

Byte-addressability v. 32-bit Data (Endianness)

Recall that accessing memory is *slow*. Therefore, rather than reading one byte at a time, we can read 32 bits (4 bytes) at a time. When we read *addr* X, what we are really doing is reading 4 bytes starting at *addr* X, giving us a 32-bit value.

Example: Load

`load x1, x0, 4` will give us address 4, 5, 6, 7 and be stored in `x1`.

Note that we don't lose byte-addressability since we can just pull the desired 4 bytes and discard the 3 we won't be reading/using.

This brings up the question of Endianness: where does the LSB go?

Definition: Little and Big Endian

In a **Little Endian** machine, the *lowest* address is at the LSB[a].
In a **Big Endian** machine, the *highest* address is at the LSB[a].

[a]LSB: Least Significant Bit.

Below is a visual of the definitions.

```
MSB           LSB
 3    2    1    0 Little Endian
 0    1    2    3 Big Endian
```

Most machines run Little Endian because it's more intuitive.

Alignment

Consider the following scenario:

```
MSB           LSB
 3    2    1   0
 7    6    5   4
```

What happens when we want to read `3, 4, 5, 6`? The common solution is to issue two loads. This situation is referred to as an *alignment issue*.

2.4 Building an ISA: Commands

Instructions in 32-bit RISC-V take the form:

`COMMAND OPERANDS`

where each instruction is fixed-size and 32-bit. Possible types of `COMMANDS` are:

(i) Arithmetic/ALU

(ii) Memory

(iii) Control-Flow

2.4.1 Arithmetic/ALU

There are two types of arithmetic operations:

(i) Operations on two registers.

(ii) Operations on one register and one immediate.

and (usually) save the result in another register (commonly referred to as the *destination register*).

Example: Source and Destination Registers

Consider the following

```
SUB x3, x2, x1
ADDI x5, x1, 10
```

We can rewrite this as `x3 = x2 - x1` and `x5 = x1 + 10` respectively. More generally, we have:

```
COMMAND DEST, SRC1, SRC2
```

There is a table of commands in [Lec. 2 Slide 48].

Aside: Shift Right Logical/Arithmetic

SRL: Shift Right *Logical* will ***not*** sign extend; i.e. will pad zeros from the left.
SRA: Shift Right *Arithmetic* ***will*** sign extend.

Example:

Consider the following:

```
addi x1, x0, 3                        x1 = 0 + 3 = 3
addi x2, x0, 5                        x2 = 0 + 5 = 5
add x3, x2, x1              x3 = x2 + x1 = 5 + 3 = 8
sra x3, x3, x1        shift x3 = 8 by x1 = 3 bits = 1
```

So the final value of `x3` is 1.

There is a table of commands in [Lec. 2 Slide 57].

2.4.2 Memory

See **2.3.3 Memory**.
There is a table of commands in [Lec. 2 Slide 63].

2.4.3 Control Flow

Definition: Program Counter/PC Register

A **Program Counter (PC/PC Register)** is a register that contains the address of the current instruction.

Control flow requires a *Program Counter* (PC) for conditional statements, function calls and returns, etc.

Jumps and Branches

Definition: Jump and Branch

A **jump** will *unconditionally* jump to a specified address.
A **branch** will *only* jump to a specified address only *if* a condition is true.

Note: We use PC-relative addressing.

Linking

Definition: Linking

Linking is the process of storing the next address *before* jumping.

We need linking to remember where we were before jumping!

Example: Sneaky Link

When we call a function `foo()`, we need to link the PC *after*[a] `foo()` so as to not get stuck in an infinite loop.

[a]the "next" PC is actually `PC + 4` since we read in 4 byte chunks!

There is a table of commands in [Lec. 2 Slide 70].

2.4.4 Pseudo Instructions

Definition: Pseudo Instruction

Pseudo Instructions are instructions that are not actually in the ISA, but can be converted into one (and are "for sanity" - Professor).

Example: `jump` and `li`

`jump label` is a pseudo instruction and is equivalent to `jal x0, label`.
`li rd, imm` is a pseudo instruction and is equivalent to `addi rd, x0, imm`.

There is a table of commands in [Lec. 2 Slide 76].

2.4.5 Calling Convention

Calling conventions are a set of rules to guarantee correctness. In RISC-V, we have three major rules:

(i) Reserved registers should be unchanged. If needed, the callee will save the register to the stack and recovers it in the end.

(ii) On call, the return must be saved on the stack.

(iii) Before returning, the frame pointer is to be recovered.

There is a table of reserved registers in [Lec. 2 Slide 83].
The caller will:

(i) Put arguments onto the stack.

(ii) Invoke the callee via the `call` (pseudo) instruction.

The callee will:

(i) Save reserved registers for the caller.

(ii) Save old based pointer.

(iii) Make room for local variables and execute the code.

(iv) Put the return value into the register.

(v) Restore the stack frame and key registers.

(vi) Return.

There is a table of caller/callee responsibilities in [Lec. 2 Slide 83].
There is a table of C types to RISC-V byte count in [Lec. 2 Slide 84].

2.4.6 Running Instructions

Machines don't understand assembly, and need an *assembler* to generate machine code to run! The *assembler* uses a table to generate the machine code.

Chapter 3

Microarchitecture

3.1 Basics

3.1.1 Reading the Table

RISC-V has 6 types of operations (R, I , S, B, U, J-type). Every instruction has an *opcode*, a 7-bit number to specify the operation of the instruction; i.e. determines the format/category of the instruction (R, I, S, B, U, J-type).

There is a table of RISC-V command types in [Lec. 3 Slide 6].