CO113 - Architecture

Prelude

The content discussed here is part of CO113 - Architecture (Computing MEng); taught by Wayne Luk, and Jana Giceva, in Imperial College London during the academic year 2018/19. The notes are written for my personal use, and have no guarantee of being correct (although I hope it is, for my own sake). This should be used in conjunction with the lecture slides, *The Hardware/Software Interface Class by Luis Ceze and Gaetano Borriello* on YouTube, and *Computer Organization and Design : The Hardware / Software Interface (Fifth Edition)* (chapters 1 to 4, and appendices B, and D), by Patterson, D., and Hennessy, J.

The second part of the course seems to be covered in sufficient detail by the YouTube playlist, which is where the majority of the information in these notes will come from.

Lecture 1 P&H 62-120

Computer architecture is a combination of ISA (instruction set architecture), and machine organisation. We can see the ISA as an interface between the high level software, and the capabilities of the physical hardware components.The benefit of having the ISA is that a piece of software can be compiled into an instruction set, and then be reused on different hardware. For example, near identical versions of the x86 instruction set are used in Intel, and AMD chips despite the two having drastically different internal designs. On the other hand, microarchitecture, or computer organisation, is the the way a given ISA is implemented in a particular processor. This comes with the additional benefit that code doesn't need to be reimplemented even if there is a drastic change in the future for the microarchitecture / machine organisation.

There are two design approaches, both of which have their benefits, and drawbacks;

- Complex Instruction Set Computers (CISC)

  The programs run on this design are closer to the high-level languages that we program in; which means that the compilers used are simpler. This is possible due to the decreasing size of transistors, and thus the increased number of gates on a chip. Programs on this instruction set tend to be smaller, as code can be represented in fewer instructions, thus saving storage.

- Reduced Instruction Set Computers (RISC)

  On the other hand, the programs running on this instruction set are closer to machine code, due to the smaller range of instructions. A more powerful, better optimised, compiler will be required. Additionally, the programs here are faster, since they have simpler instructions - but they may require more instructions to achieve what a CISC can do in one, thus there may be a trade-off. It's also easier to build a chip with less instructions, which leads to lower development costs. Due to the smaller physical size of the chips, we can not only fit multiple chips together, but also use the space for memory, since accessing memory outside of the chip is very slow (compared to the high-speed registers nearby).

In this course, we will be working mostly on a MIPS processor. Generally, the instructions consist of an opcode, which is what it does, and an operand (which includes the registers, memory locations, and data). This should be fairly similar to the very end of **CO112 - Hardware**. The design principle for RISCs is that the processor should have good performance, and be relatively simple to implement. In MIPS, there are 3 main types of instructions; R (register), I (immediate), and J (jump), all of which have a fixed size of 32 bits.

MIPS is representative of modern RISC architectures, and has 32 registers, each being able to store 32-bit data. The registers are named \$0..\$31, with \$0 being typically wired to ground (logic 0), and the others being used for general-purpose storage. MIPS is known as a register-register, or load-store architecture, which means that there are two different sets of instructions; one that is extremely fast, and works between registers, and another set working with memory access, which tends to be slower. The goal is to minimise memory access, as accessing data from memory tends to be much slower than accessing memory located in the registers on the chip. Here are some examples of these instructions;

- register-register

  `add $1, $2, $3` reg1 = reg2 + reg3

- load-store

  `lw $8, Astart($19)` reg8 = M[Astart + reg19]

R-type instructions can be used for arithmetic, comparisons, logical operations, etc. and have a general format as follows (the example describes `add $8, $17, $18`). It's important to note that we have an additional 6 bits at the end for the function, since having a 6-bit opcode only leaves us 64 (2^6) instructions, which is quite limited even for a RISC instruction set. In addition, the shift amount specifies the amount of bits to shift, if it was a shift instruction, however it's redundant in this case;

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
0	17	18	8	0	32
opcode	source 1	source 2	destination	shift	function

I-type instructions are used for memory access, conditional branching, or arithmetic with constants. An example of doing addition with constants is `addi $1, $2, 100`, which does reg1 = reg2 + 100. The example displayed below is `lw $8, Astart($19)`, which does reg8 = M[Astart + reg19].

6 bits	5 bits	5 bits	16 bits
35	19	8	`Astart`
opcode	source	destination	immediate constant

Finally J-type instructions are jump to instructions in memory, for example, `j 1236` would be an unconditional jump to the instruction at address 1236. An unconditional jump has the following format;

6 bits	26 bits
2	1236
opcode	memory location

However, we can also have jump instructions, which are I-type, or R-type, for example `bne $19, $20, Label` is an I-type instruction, where the program jumps to `Label` if registers 19, and 20 aren't equal. An R-type example would be `jr $ra`, where it jumps to the address in register ra.
Consider the following program, and its equivalent in machine code, the registers are labeled in alphabetical order (reg16 = f, reg20 = j, etc);

```
if (i == j) {
  f = g + h;
} else {
  f = g - h;
}

      bne $19, $20, Else # if i ≠ j goto Else
      add $16, $17, $18  # f = g + h
      j   Exit           # goto Exit
Else: sub $16, $17, $18  # f = g - h
Exit:
```

Since we only have two types of conditional branches, `bne`, and `beq`, we need `slt`, which does the following - `slt $1, $16, $17`, if reg16 < reg17, then it sets reg1 to 1, otherwise it's set to 0. Then, we can use `bne`, with `$0`, since reg0 is always set to logic 0.

Lecture 2 P&H 28-53

One of the questions raised in this lecture is the following; "Is a 20% cheaper processor, with the same performance good enough?". While this may seem straightforward, from a consumer's perspective, it's important to note that a consumer has instant gratification from buying a product, but developing one would take time. In this time, competitors are also trying to improve on their product, and as such you can't just know the price, and performance of a competitor's product **now**, but you also need to predict the improvement.

CPI is the **average** number of clock cycles required per instruction. Note that it's the average, because some instructions may take more cycles to complete. For a given program P, we can get the number of cycles required for P by doing the number of instructions in P, multiplied by the CPI. The execution time for P is the number of cycles in P, multiplied by the clock cycle time (which is $\frac{1}{\text{clock speed}}$). Assuming that for a set of programs P_1, ..., P_n, the workload is equal, we can calculate the average execution time for the set by taking the mean of the execution times.

Example

Consider two machines, M_1, and M_2, which implement the same instruction set that has 2 classes of instructions; A, and B. The CPI for M_1 on class A is A_1, B, is B_1, and the same for M_2. The clock speed of M_1 is C_1 MHz, and similar for M_2. If we were to compare their peak and average performance of N instructions, half of which are of class A, and the other half of class B, we'd need to find the ratio of execution times.

In order to find the peak performance of N instructions for M_1 (let it be P_{P1}), we take the clock cycle time (which is $\frac{1}{C_1}$, multiply it by the number of instructions N, miltiply it by the **minimum** CPI for M_1 (which would be min(A_1, B_1)), we'd get $\frac{N(\min(A_1,B_1))}{C_1}$. To compare the two, we take $\frac{P_{P1}}{P_{P2}} = \frac{\min(A_1,B_1)\cdot C_2}{\min(A_2,B_2)\cdot C_1}$.

We do a similar process for finding the average performance, let it be P_{A1}, but instead of multiplying it by the minimum CPI, we take the average, hence we multiply by $\frac{A_1+B1}{2}$. To compare the two, we take $\frac{P_{A1}}{P_{A2}} = \frac{(A_1+B_1)\cdot C_2}{(A_2+B_2)\cdot C_1}$.

Our goal is to minimize the execution time, which is to minimise instruction count $\times$ CPI $\times$ cycle time. Consider this example, comparing SUN 68000, and their newer SUN RISC. In the RISC device, there are 25% more instructions, and the cycle time is

	SUN 68000	SUN RISC
Instruction Count Ratio	1.0	1.25
Cycle time	40ns	60ns
CPI	5.0 - 7.0	1.3 - 1.7
Execution Time Ratio	2	1
Price Ratio	1	1.1 - 1.2