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 50% longer. However, the CPI is much lower, as the instructions are simpler, thus requiring less cycles. The price has increased, but the performance has doubled.

	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

The processor time is measured by the seconds per program, which is calculated as follows; $\frac{\text{time}}{\text{program}} = \frac{\text{instructions}}{\text{program}} \cdot \frac{\text{cycles}}{\text{instruction}} \cdot \frac{\text{time}}{\text{cycle}}$.

RISC

Regarding the principles of RISC instruction set design, the common cases should be optimised, thus reducing the CPI. A small number of general purpose registers (32 in MIPS), simplifies things, and allows the design to be more adaptable to new technologies. The smaller chip size allows for a higher yield, thus reducing the cost of production. On the other hand, the lower number of instructions increases the code size, and smarter compilers are needed, since the instructions are further away from the software level than with a CISC instruction set.

Performance Trends

In 2004, the trend in power usage hit a peak, due to heat not being able to be removed from the chip at a reasonable rate. The voltage also cannot be reduced further, which is why the trends seemed to have become flat. $P = C \cdot V^2 \cdot F$, where P is power, C is capacitive load, V is voltage, and F is frequency.

Other than just increasing clock speed, performance can be increased in other ways; including faster local storage, concurrent execution, and newer technologies. Implementning on-chip caches allows for faster execution due to the faster memory closer to the chip, which would be a significant improvement compared to fetching from RAM. Concurrent execution can be achieved by multiple function units (super scalar), a pipeline execution, or multiple instruction streams (multi-threading). Newer technologies, such as GPUs can also be used for specialised loads.

Benchmarking

There are a number of ways of benchmarking, each with their benefits, and drawbacks as follows;

method	pros	cons
actual target workload	representative	very specific, not portable difficult to measure hard to identify problems
full benchmarks	portable widespread usage	less representative
kernel benchmarks	easy to use used early in design cycle identify peak performance	peak is not representative

Lecture 3

Considering the software side of parallelism; we have parallel requests, parallel threads, parallel instructions, and parallel data. Parallel threads schedule tasks; for example if you have an instruction that takes longer to process since it has to read from main memory, or wait for another resource, another task can be scheduled to run during this time. Since a processor core has multiple functional units, instructions can be arranged in a pipeline, where different stages are processed at the same time. Finally, data can be parallelised, as each item of data can contain multiple chunks of data, each of which can be operated on separately.

In MIPS, we have 3 different types of addressing;

- register addressing — accessing the data in registers
- immediate addressing — data is contained within the instruction (I-type)
- base addressing — accessing data in memory with load/store instructions
- PC-relative addressing — replaces the register with the program counter (in the I-type load)

We can classify architectures by how they address temporary storage. Here we cover three main types - all of which are operating on the same code; which is `C = A + B`;

- stack — operands are implicitly specified at the top of the stack

  `push A; push B; add; pop C`

  this adds the top pair of items on the stack

  pros: it has a simple evaluation model, and the code is dense

  cons: this model is less flexible, has no random access, and is slow if the stack is in memory

- accumulator — one operand in the accumulator

  `load A; add B; store C`

  this adds the accumulator, and the data in memory

  pros: there is minimal internal storage, and has short instructions

  cons: there is frequent memory access, therefore it is slower

- register — we explicitly state the operands

  `load R1 A; add R2, R1, B; store C, R2`

  this simply adds two registers

  pros: this is the general model for code generation, and has faster register access

  cons: this requires you to name all the operands, and also has longer instructions

Most modern architectures are register based, as it's still faster, as there is less memory traffic, as well as the code being denser. At the start, the first computers used single accumulators, as memory was still expensive, and therefore registers had to be used sparingly.

Amdahl's Law

When some instructions are used frequently, and are normally expensive to compute, there are three possible approaches (for example, repeatedly calculating $x^2 + y^2$);

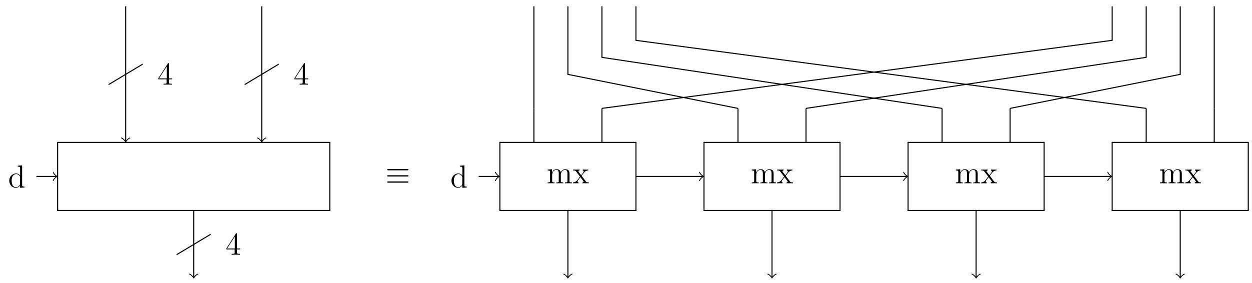
1. add instruciton, accumulator, or load-store
2. add, and square instructions, accumulator, or load-store
3. custom sumsq instruction, with a dedicated circuit

However, this is not always beneficial (or worth the additional cost, and time). For example, consider a program that takes T_{old} time to run, and a fraction of the code α can be sped up β times. Now, we can calculate the new runtime of the code as $T_{\text{new}} = \alpha\frac{T_{\text{old}}}{\beta} + (1-\alpha)T_{\text{old}}$. Let's have an example, wehre 90% of the code can be sped up 100 times, such that $\alpha = 0.9$, and $\beta = 100$. By running this calculation, we can say that $T_{\text{old}} \approx 9.17 \cdot T_{\text{new}}$ - the code is less than 10 times faster.

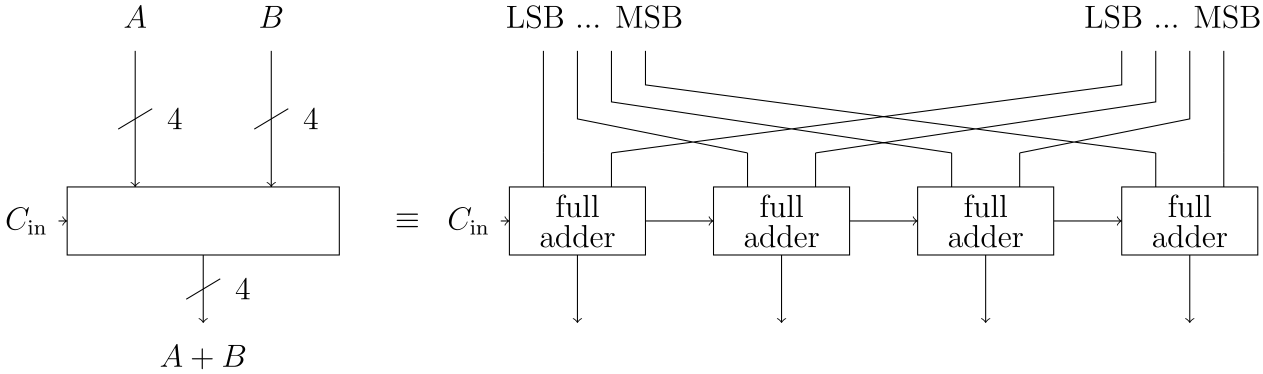
Lecture 4

There are two ways of representing negative numbers in binary, two's complement, and sign-and-magnitude. When we use sign-and-magnitude, it may be more intuitive for us, but for a computer to do addition on it may be problematic as we can easily lose (or gain) the sign bit. On the other hand, two's complement is more complex, but allows for easier operations. For example, you can repeat the most significant bit (e.g. $\texttt{10}_{\text{2C}} = \texttt{111110}_{\text{2C}} = -2_{\text{Dec}}$)

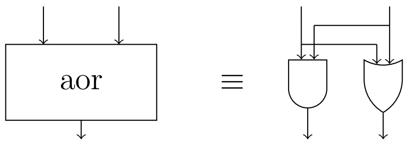
The layout of a MIPS ALU is similar to the basic one covered in **CO112**, as in, it has separate units for bit-wise AND, bit-wise OR, addition, etc. and also does the all the operations, then selects one based on the input. Similarly, it also uses the same slash notation to denote n lines being connected. On the gate level, it's important to remember the following;



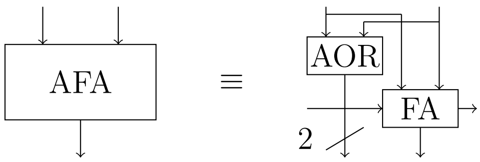
A similar diagram can also be used for the ripple carry adder, which joins n full adders, to create an n-bit ripple adder.



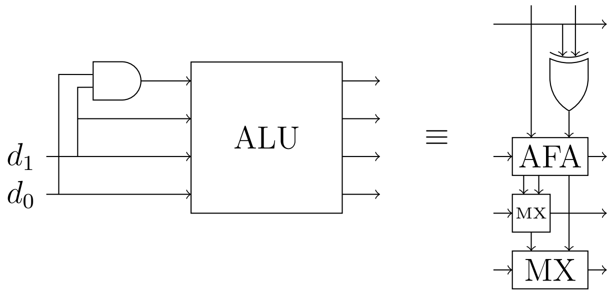
With this full adder, we are able to use this as as subtractor. For example, when working with $A - B$; take the ones complement of B (which is inverting B bitwise), and add it to A, and setting C_{in} to 1. Bitwise inversion is done with an XOR, when the otger input is 1. Other important circuits found in our ALU, are the AOR (which is shown below);



This can then be combined with a single full adder block, to create AFA;



With these components, we can build our first ALU block;



This design for the ALU has the functions $d_0 d_1$, where 00 is AND, 01 is OR, 10 is addition, and 11 is subtraction. Remember that the carry is only set to 1 at the start when we're doing subtraction. The top line is also set to 1 when we're doing subtraction.

The carry path is often the slowest line, as it needs to go through many gates of logic, which limits the clock rate, since clock rate $\approx \frac{1}{\text{delay of slowest path}}$, given we have an edge-triggered design, and a few other factors (from P&H Appendix B. 11)

Lecture 5 — P&H 183-196

Multiplication Algorithm

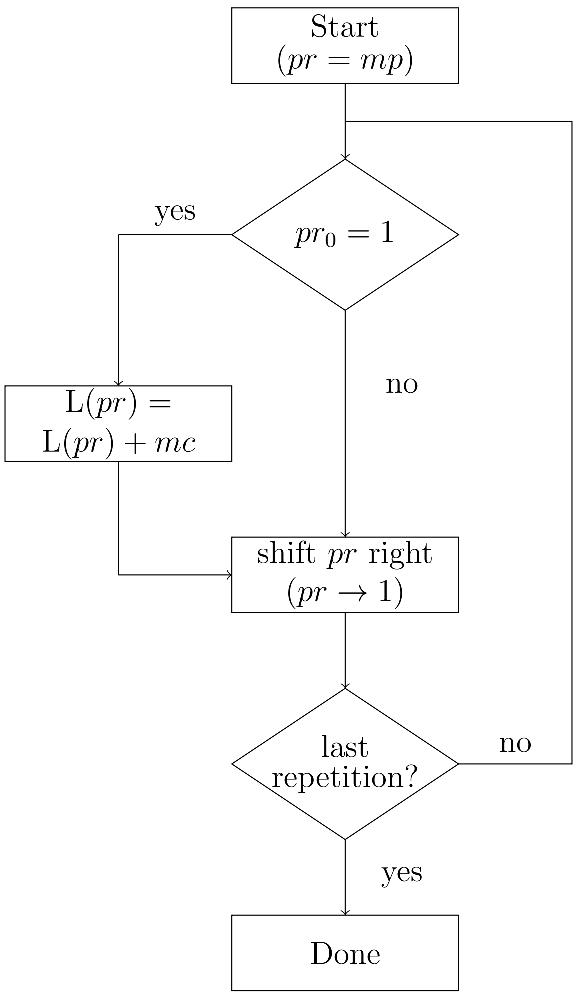
Consider the example $2 \cdot 11 = 22$, we have the multiplicand times the multiplier = product. However, to do this bitwise, we have to do the following (let $c \leftarrow n$ mean c shifted n bits to the left, and b_i mean the i^{th} bit of b);

$$
\begin{array}{cccccccc|l}
 & & & & 0 & 0 & 1 & 0 & \text{multiplicand } (c) \\
\times & & & & 1 & 0 & 1 & 1 & \text{multiplier } (p) \\
\hline
 & & & & 0 & 0 & 1 & 0 & (c \leftarrow 0) \cdot p_0 \\
 & & & 0 & 0 & 1 & 0 & & (c \leftarrow 1) \cdot p_1 \\
 & & 0 & 0 & 0 & 0 & & & (c \leftarrow 2) \cdot p_2 \\
+ & 0 & 0 & 1 & 0 & & & & (c \leftarrow 3) \cdot p_3 \\
\hline
 & 0 & 0 & 1 & 0 & 1 & 1 & 0 &
\end{array}
$$

Note that the third line is all 0s, because p_2 is 0, and multiplication is just AND. The idea is that the product is the miltiplicand shifted succesively by 1 bit relative to the multiplier; CSAA - conditional shift and add. We only really need to shift when the bit isn't 0.

Another Multiplication Algorithm

However, there are other options for multiplication algorithms, which can save silicon space; we can use a 32-bit ALU, and a 64-bit register, which stores both the product, and the multiplier initially.



Booth's Algorithm

When we have a string of repeated 1s, we can change n additions into 1 addition, and 1 subtraction. As we're summing a geometric series, when we do repeated additions, such that $m + 2m + 2^2m + ... + 2^{k-1}m =$

$-m + 2^k m$. This is much easier to compute, as all we have to do is to do an arithmetic shift on m, and a subtraction. However, instead of checking only the LSB (pr_0), we also check the previous LSB, let it be pr_{-1}. We have the following cases, written $pr_0 pr_{-1}$; 00 or 11 - we're in the middle of a string of 0s (or 1s, respectively), no action is needed, 01 - we're at the end of a string of 1s, $pr = pr + mc$ (where mc is the shifted multiplicand), and 10 - we're at the start of a string of 1s, $pr = pr - mc$. Note that in my version of the slides (2018 - 2019 academic year), there is a typo on slide 15. The "corrected" version is below (this is working on $\texttt{0010}_2 \times \texttt{0110}_2$), and $mc$is `0010`. Also note that (pr) means the left half of the product register;

iteration	original		Booth's	
	step	product	step	product
0	initial values	0000 011**0**	initial values	0000 011**0** **0**
1	1a: 0 - no operation	0000 0110	1a: 00 - no operation	0000 0110 0
	2: product shift right	0000 001**1**	2: product shift right	0000 001**1** **0**
2	1b: 1 - L(pr) = L(pr) + mc	0010 0011	1c: 10 - L(pr) = L(pr) − mc	1110 0011 0
	2: product shift right	0001 000**1**	2: product shift right	1111 000**1** **1**
3	1b: 1 - L(pr) = L(pr) + mc	0011 0001	1d: 11 - no operation	1111 0001 1
	1: product shift right	0001 100**0**	2: product shift right	1111 100**0** **1**
4	1a: 0 - no operation	0001 1000	1b: 01 - L(pr) = L(pr) + mc	0001 1000 1
	1: product shift right	0000 1100	2: product shift right	0000 1100 0

Division

This algorithm was invented by Briggs; dividend = quotient × divisor + remainder. We can work through an example of the first algorithm as follows; case 2b is when $rem < 0$, and 2a is when $rem \geq 0$. Note that SLL means we are doing a logical left shift on the quotient,and SR means we are shifting the divisor to the right. This is working through $\frac{7}{2}$;

iteration	step	quotient	divisor	remainder
0	initial values	0000	0010 0000	0000 0111
1	1: $rem = rem - div$	0000	0010 0000	1110 0111
	2b: $rem = rem + div$; SLL; $Q_0 = 0$	0000	0010 0000	0000 0111
	c: SR	0000	0001 0000	0000 0111
2	1: $rem = rem - div$	0000	0001 0000	1111 0111
	2b: $rem = rem + div$; SLL; $Q_0 = 0$	0000	0001 0000	0000 0111
	c: SR	0000	0000 1000	0000 0111
3	1: $rem = rem - div$	0000	0000 1000	1111 1111
	2b: $rem = rem + div$; SLL; $Q_0 = 0$	0000	0000 1000	0000 0111
	c: SR	0000	0000 0100	0000 0111
4	1: $rem = rem - div$	0000	0000 0100	0000 0011
	2a: SLL; $Q_0 = 1$	0001	0000 0100	0000 0011
	c: SR	0001	0000 0010	0000 0011
5	1: $rem = rem - div$	0001	0000 0010	0000 0001
	2a: SLL; $Q_0 = 1$	0011	0000 0010	0000 0001
	c: SR	0011	0000 0001	0000 0001