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);

```
1 if (i == j) {
2   f = g + h;
3 } else {
4   f = g - h;
5 }
6
7       bne $19, $20, Else # if i ≠ j goto Else
```

```
8         add $16, $17, $18  # f = g + h
9         j   Exit           # goto Exit
10 Else:  sub $16, $17, $18  # f = g - h
11 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 × CPI × 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 $x^2 + y^2$);

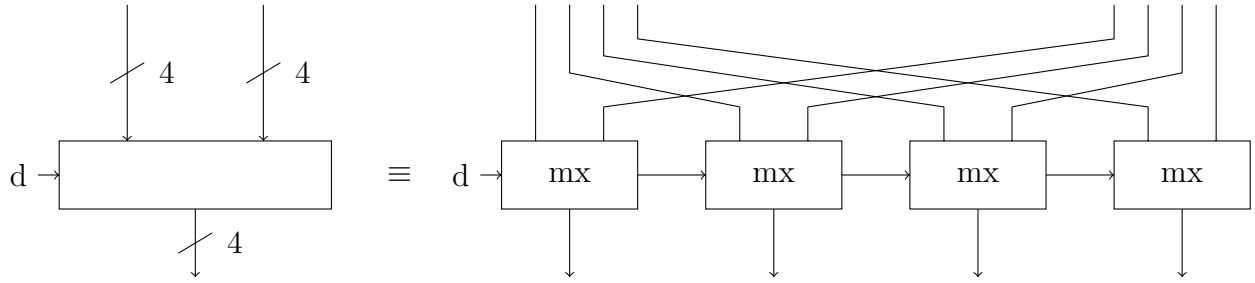
1. add instruciton, accumulator, or load-store
2. add, and square instructions, accumulator, or load-store
3. custom sumsq instruction, with a dedicated circuit

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.

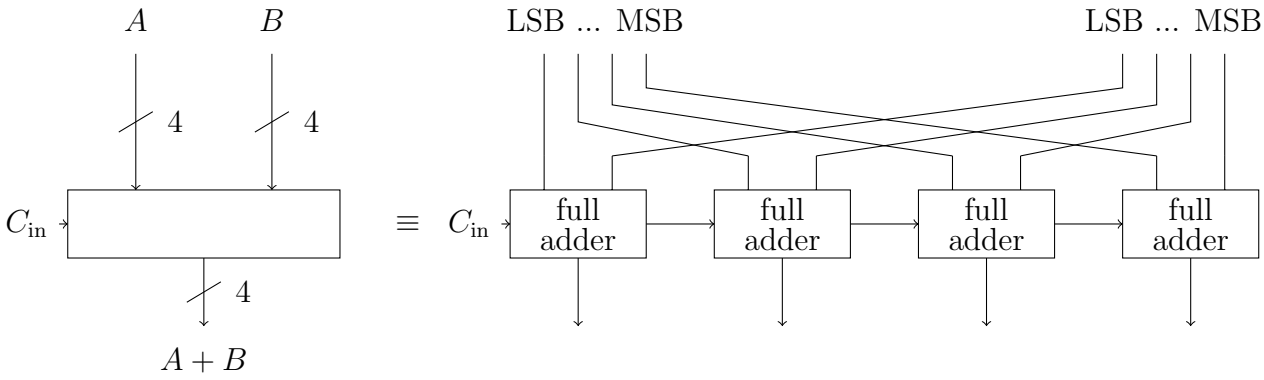
Lecture 4

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. $\texttt{10}_{\text{2C}} = \texttt{111110}_{\text{2C}} = -2_{\text{Dec}}$)

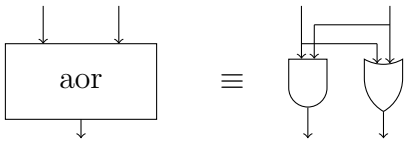
The layout of a MIPS ALU is similar to the basic one covered in **CO112**, as in, it has separate units for bit-wise AND, bit-wise OR, addition, etc. and also does the all the operations, then selects one based on the input. Similarly, it also uses the same slash notation to denote n lines being connected.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



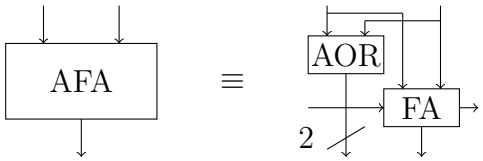
A similar diagram can also be used for the ripple carry adder, which joins n full adders, to create an n-bit ripple adder.



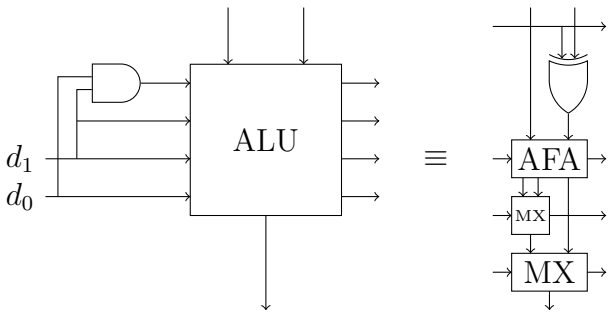
With this full adder, we are able to use this as as subtractor. For example, when working with $A - B$; take the ones complement of B (which is inverting B bitwise), and add it to A, and setting C_{in} to 1. Bitwise inversion is done with an XOR, when the otger input is 1. Other important circuits found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

Multiplication Algorithm

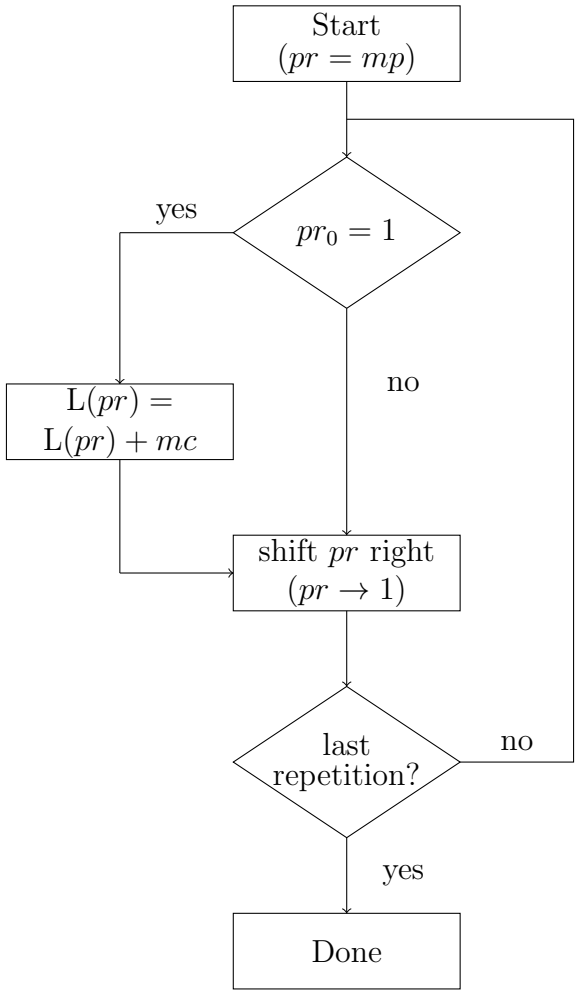
Consider the example $2 \cdot 11 = 22$, we have the multiplicand times the multiplier = product. However, to do this bitwise, we have to do the following (let $c \leftarrow n$ mean c shifted n bits to the left, and b_i mean the i^{th} bit of b);

				0	0	1	0	multiplicand (c)
$\times$				1	0	1	1	multiplier (p)
				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				$(c \leftarrow 3) \cdot p_3$
	0	0	1	0	1	1	0	

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

Similar to multiplication, we can refine our implementation of division by replacing the divisor shift to the right, with a remainder shift to the left. By doing this, we can reduce the 64-bit ALU to 32 bits. As we are shifting the remainder to the left, and we're doing the same shift to the quotient, we can combine the registers like before.

Lecture 6 — P&H 244-272

Seriously, for this lecture, just look at the slides. There's too many diagrams for me to draw in TikZ.

In general, the control unit is just a combinatorial unit, which takes in the 6 bits from the opcode, and has a 9-bit output, which controls the multiplexers, ALU, and the read / write operations. The initial implementation was having separate datapaths for the different types of instructions; register-based, memory-based, and branch. This can then be combined with multiplexors, which allow the right blocks to be conected, and finally the control unit unifies it by activating relavent parts of the combined datapath, based on the instruciton.

Note that often the addresses for instructions will increment in 4, since memory is normally byte addressable, and the instructions we are working with are 32-bit. The design in the slides abstract the circuit into a single cycle data path, but it's important to note that it isn't the case, especially due to memory access as that would require more cycles.

Lecture 7

The following accesses are needed in the execution cycle;

type	instruction fetch	read register	ALU operation	load / store data	write to register
R-type	✓	✓	✓		✓
load	✓	✓	✓	✓	✓
store	✓	✓	✓	✓	
branch	✓	✓	✓		
jump	✓				

From the above, you will notice that some operations take multiple stages to do, such as load taking all 5, and jump taking only 1. Due to the single clock cycle data path design, all of them take one cycle to finish, regardless of the number of steps. As clocks have a fixed tick time, jump will still take the same amount of time as load, even though it's a much faster instruction.

Multi-cycle datapath

This comes with multiple advantages; we're likely to have shorter cycles, but will need more of them (for example, R-type instructions would need 4 cycles to complete, and jump would only need 1). We can also combine memory together, such that instruction, and data are stored in the same location. The ALU can also be reused, but we'd also need the IR, which stores the instruction. More registers will be needed to save the state, leading to a more complex control unit.

In order to build this, we'd need new internal registers; IR, A, B, ALU_{out}, MDR.

For example, when working on the load instruction, which has the effect;

$$\text{Reg}[\underbrace{\text{dest}}_{\text{IR}_{20-16}}] = \text{M}[\text{Reg}[\underbrace{\text{source}}_{\text{IR}_{25-21}}] + \text{sign-ext}(\underbrace{\text{addr}}_{\text{IR}_{15-0}})]$$

This instruction can be broken down into smaller steps, as follows, which are RTL (Register Transfer Level) descriptions;

cycle 1: IR = M[PC], PC = PC + 4

cycle 2: A = Reg[source]

cycle 3: ALU_{out} = A + sign-ext(addr)

the sign for addr needs to be extended to a 32-bit number, as the ALU is 32-bit

cycle 4: MDR = M[ALU_{out}]

cycle 5: Reg[dest] = MDR

We can tabulate all the execution steps as the following;

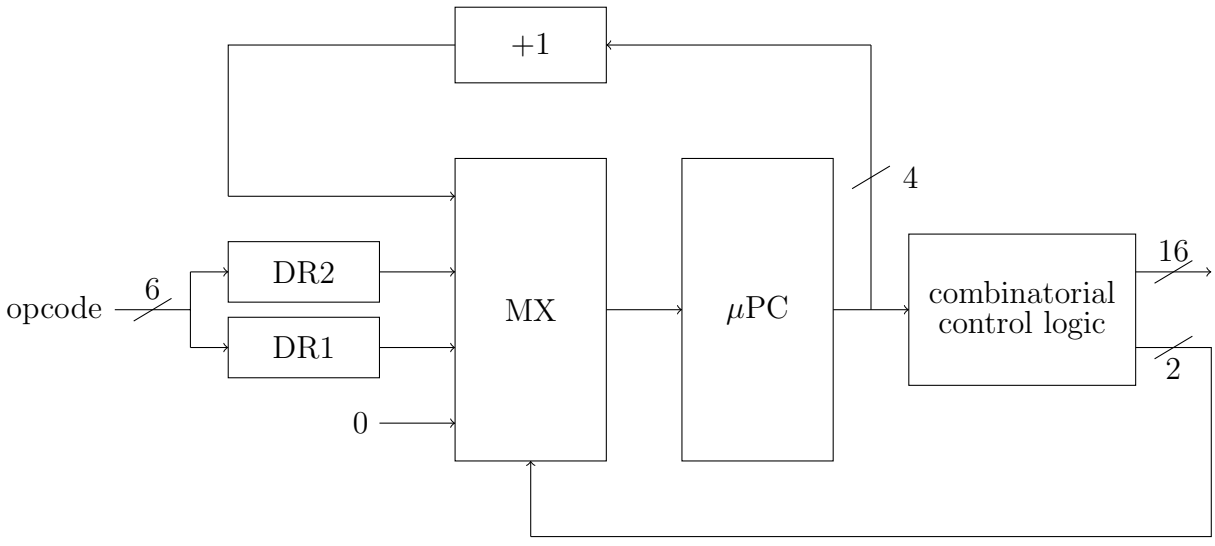
Step	R-type	memory-reference	branches	jumps
Instruction fetch	IR = M[PC] PC = PC + 4 S_0			
Instruction decode or register fetch	A = Reg[$IR_{25\text{-}21}$] B = Reg[$IR_{20\text{-}16}$] ALU_{out} = PC + (sign-extend($IR_{15\text{-}0}$) << 2) S_1			
Execution, address computation, branch or jump completion	ALU_{out} = A op B S_6	ALU_{out} = A + sign-extend($IR_{15\text{-}0}$) S_2	if (A == B) then PC = ALU_{out} S_8	PC = PC[$IR_{31\text{-}28}$] || ($IR_{25\text{-}0}$ << 2) S_9
Memory access or R-type completion	Reg[$IR_{15\text{-}11}$] = ALU_{out} S_7	load: MDR = M[ALU_{out}] S_3 store: M[ALU_{out}] = B S_5		
Memory read completion		load: Reg[$IR_{20\text{-}16}$] = MDR S_4		

Once again, just like in **CO112**, this can be modelled as a state transition diagram, with the input being determined by the control unit.

Lecture 8 — P&H C.3-C.6

One of the main issues with our representation of the control unit output logic as a direct FSM is the size, which would have be around $2^10 \cdot 20$, which is roughly 20.5Kbits. Another issue is the readability of it all; while it's very sequential (and we can exploit that later), especially without grouping the outputs. Hence if we were to define fields, we can have each one correspond to one, or more, control signal(s) to acheive a task. For example, if we were to define SRC1 as a field, we could say that SRC1 = A means that ALUSrcA = 1. By assigning values to a field, which represents a control signal assignment, we reach a higher level of abstraction (further than listing the indvidual signals in the state diagram). Another example is having $ALU_{control}$ be Add, Fn, or Sub, with the codes 00, 10, or 01 respectively. We can tabulate this as such;

state	label	ALU control	SRC1	SRC2	memory	reg. control	PC write control	sequencing
0	Fetch	Add	PC	4	ReadPC		ALU	Seq
1		Add	PC	ExtShft		Read		Dispatch 1
2	Mem1	Add	A	Extend				Dispatch 2
3	LW2				ReadALU			Seq
4						WriteMDR		Fetch
5	SW2				WriteALU			Fetch
6	Rformat1	Fn	A	B				Seq
7						WriteALU		Fetch
8	BEQ1	Sub	A	B			ALU outcond	Fetch
9	JUMP1		A	B			Jump addr	Fetch



Note that we'll still have to define DR1, and DR2, the dispatch ROMs. However, the total size of this is around 0.8Kbits, compared to the 20.5Kbits we had before. We could also perform vertical encoding on the output logic, which would reduce the control logic signals, but this comes with its own drawbacks. Horizontal microencoding (which is what we have now) exploits parallelism, and has very little control overhead, however it requires a larger ROM. On the other hand, vertical encoding can be slow, due to the decoding delay.

Lecture 9

Note that pipelining is not examined. In order to handle unexpected events, the processor has to implement addtional circuitry. There are two types of unexpected events that can occur, which both have to be handled with extreme urgency;

- interruptions

  these are external events that disrupt the execution cycle, such as input or output, like the power button being pressed

- exceptions

  these are unexpected events from a program, such as an invalid opcode, or integer overflows

In the event of an emergency, the program has to do the following;

1. find out the cause

   in order to handle the error, the processor will have to find out the reason

   this is done by having an additional register, the Cause register, which is also 32 bits

2. return to normal execution (sometimes)

   in order to retain the program execution point, we need to store the restart address in the exception program counter (EPC)

   the control is then transferred to the OS by executing instructions run at the exception address, which is a constant wired into the PC multiplexer

To implement this, we also need to modify the finite state machines to have additional states for each error (see the lecture slides for the example).

Interlude and Introduction

As the YouTube playlist covers most of the content, I will be using that as the main point of reference, and not Panopto.

While we'd rather write Java, or C, because it's more human friendly, hardware requires strings of bytes which are voltage highs, and lows. While machine code was fine at the start, when machines became more complex, and we could no longer keep up. At this point, we were able to use assembly, which would have equivalent instructions in machine code, but were still human readable. After this, we have an even higher level of abstraction, where a single line of a high-level language, like C, or Java, can be compiled into many lines of assembly. The current lifetime of a program is having the user written program in a high level language, compiled to assembly, which is then compiled to machine code, and run on the hardware.

During the compilation, the compiler can run optimisations on the code, allowing it to be more efficient to some extent, without the programmer having to do so.

Memory & Data

Memory, data, and Addressing

Data needs to be moved from memory to registers, in order for the CPU to operate on it (mentioned in the first part of the module). There's also an instruction cache on the CPU that holds recently used instructions, such as loops - all handled by hardware.

The bandwidth between the memory, and CPU can limit performance (bottleneck). There are two ways to improve performance; either improving the hardware, or by having larger memory in the chip (cache).

The transition between voltages can limit the speed of our processor, as we don't want values within the intdeterminate range (between 0.5v, and 2.8v - arbitrary).

Bits, Bytes, and Words

In binary, a byte is 8 bits. Converting between hexadecimal, binary, and decimal should be trivial. A byte is two hex digits, hex numbers are represented in C, as `0xFFF3FF1F` etc. Memory is also byte addressable, and normally an OS provides an **address space** which is private to each process. A program can modify data in its own state, but not others.

Each machine has a **word size**, which is the nominal size of integers in a given machine. Older machines ran on 32-bit words, which limited address to 4GB, but that was too small for intensive applications. However, most modern x86 systems are now on 64-bit words, which allows for around 18 exabytes of memory. In order to group words, we address the word by the address of the first byte, for example, the address of the second word in a 64-bit machine would be 0008_{10}.

Memory Addresses

An address is a location in memory, a pointer is a data object which contains an address. These are the sizes of objects, in bytes;

Java	C	32-bit	x86-64
boolean	bool	1	1
byte	char	1	1
char		2	2
short	short int	2	2
int	int	4	4
float	float	4	4
	long int	4	8
double	double	8	8
long	long long	8	8
	long double	8	16
(reference)	pointer *	4	8

In big endian notation, the MSB has the lowest address, whereas in little endian, LSB has the lowest address. Little endian is used in x86, which is what we'll be using.

Addressess, and Pointers in C

To be completely honest, this isn't all that useful for this module. But it helps build some understanding.

```
1 int x, y; /* finds two locations in memory to store 2 integers */
2 int *ptr; /* declares a variable ptr, which points to an integer data item */
3 ptr = &x; /* assigns ptr to point to the address of x */
4 y = *ptr + 1; /* same as y = x + 1; */
```

Arrays

Arrays are adjacent locations in memory, which store the same type.

```
int big_array[128]; /* allocates 512 adjacent bytes in memory (128 * 4) */
int *array_ptr;
array_ptr = &big_array[i];
array_ptr = &big_array[0] + i*sizeof(big_array[0]); /* same as above line */
```

C-style strings are represented as an array of bytes, with a null terminator, which is just a byte of 0s. In order to compute the length of this string, we'd count up until we reach the null terminator.

Boolean Algebra, and Bit Manipulation

We really could just skip this, since it's fully covered in **CO112**, and **CO140**. The same bit vector operations can be done on any integeral data type in C (`long`, `int`, `short`, and `char`).

```
p && *p++; /* avoids null pointers, as 0 is false */
/* short for the below */
if (p) {
  *p++;
}
```

Bit vectors can be used to represent sets, when we have a w-bit vector, representing a set $A = \{0, ..., w-1\}$, such that bit $a_j == 1 \leftrightarrow j \in A$. This way, we can do bitwise operations, such as doing intersection, union, symmetric difference, and complement.

Numbers

Binary Encoding

Consider a deck of cards; we have four approaches;

1. use 52 bits

   this is a one-hot (one bit set to 1)

2. use 4 bits, and 13 bits

   this is a two-hot, where the first four bits represents the suit, and the last 13 represent the card

3. use 6 bits

   this method is done by storing a number in binary, up to 52

4. use 6 bits

   use 2 bits to store suit, and the remaining 4 to store the value

   we can get the suit with a bitwise mask of `0x30`, and similarly use bitmask `0x0F` to get the value

Integer Encoding

We can represent an n-bit binary digit as $\sum_{i=0}^{n-1} 2^i b_i$, and therefore the biggest number we can represent in an unsigned n-bit binary digit is $2^n - 1$. Binary addition, and subtraction is covered here as well, but we can refer back to **CO112**.

However, this representation doesn't allow us to represent negative numbers. This leads to two possible approaches; sign-and-magnitude, and twos complement. The former has an issue in `0x80`, which is -0, and `0x00`, which is 0 (positive), since we're taking the first bit as the sign, and the remaining 7 to represent the magnitude - which also leads to issues with arithmetic. On the other hand, the latter negates the MSB, such that we can represent an n-bit twos complement digit as;

$(\sum_{i=0}^{n-2} 2^i b_i) - 2^{n-1} b_{n-1}$.

This has significant benefits, as we can now easily do arithmetic on it, since we have $\texttt{1111}_2 = -1_{10}$, and $\texttt{0000}_2 = 0_{10}$. It also simplifies hardware, as our adders would work for both unsigned, and twos complement integers. In order to negate an unsigned integer, let it be `x`, we take the complement of `x`, and add 1 to it. As we still have limits for both signed, and unsigned numbers, the CPU may throw an exception for signed values, but most won't for unsigned values, leading to overflow (or underflow). With a word size w, the unsigned values exist within the range $[0, 2^w - 1]$, and twos complement values exist within the range $[-2^{w-1}, 2^{w-1} - 1]$

Integers in C

The limits vary from machine to machine, as a 64-bit machine would have a much higher range. By default, C uses signed integers, so we'd have to add the `U` suffix to force unsigned. If signed values, and unsigned values are used in the same expression, the signed value will be implictly cast to an unsigned value; therefore `-1 < 0U` would evaluate to false.

Bit Shifting, and Sign Extension

These are the shift operations on unsigned integers;

expression	binary	denary
`x`	`00000110`	6
`x << 3`	`00110000`	48
`x >> 2`	`00000001`	1
`y`	`11110010`	242
`y << 3`	`10010000`	144
`y >> 2`	`00111100`	60

On the other hand, with signed (twos complement) integers, we have to deal with the difference between an arithmetic shift, and a logical shift. The arithmetic shift fills with the most significant bit on the left, which maintains the sign, whereas the logical shift just fills with 0s regardless;

expression	binary	denary
`x`	`01100010`	98
`x << 3`	`00010000`	26
`x >> 2` (logical)	`00011000`	24
`x >> 2` (arithmetic)	`00011000`	24
`y`	`10100010`	-94
`y << 3`	`00010000`	16
`y >> 2` (logical)	`00101000`	40
`y >> 2` (arithmetic)	`11101000`	-24

We can use this method to get the second most significant byte of an integer as follows;

expression	binary
`x`	`01100001 01100010 01100011 01100100`
`x >> 16`	`00000000 00000000 01100001 01100010`
`(x >> 16) & 0xFF`	`00000000 00000000 00000000 01100010`

The same method can be used to extract the signed bit of a signed integer as follows; given a 32-bit signed integer `x`, we can do `(x >> 31) & 1`.

In order to extend the sign of an integer, we simply repeat the most significant bit, as this maintains the value (and of course, the sign).

Fractional Binary Numbers

Consider the fractional binary number $\texttt{10111.101}_2$, which is done as;

16	8	4	2	1	.	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$
1	0	1	1	1	.	1	0	1

This representation of binary numbers is also able to use the shifts for multiplication (and therefore division) by powers of 2. However, this is a fixed point representation. The closer the fixed point is closer to the MSB, the lower the range, but the higher the precision, and on the other hand, if the fixed point is closer to the LSB, we have a higher range but low precision.

Floating Point

The IEEE floating point standard (754) is analogous to scientific notation, and is supported by all major CPUs today. This standard was driven by the requirement for having standards to handle rounding, and over/underflow. It's hard to make fast in hardwae, but behaves well numerically. Given some value V, in base 10, we have $V_{10} = (-1)^s \cdot M \cdot 2^E$, where s is the sign bit, the significand (or mantissa) M, being a fractional value in the range [1.0, 2), and an exponent E, which weights the value by a power of 2. In memory, it's represented as a sign bit, an 8-bit (11 in 64-bit) `exp` field encoding E (not equal), and a 23-bit (52 in 64-bit) `frac` field encoding M, also not equal. Single precision is 32-bit, and double is 64-bit. The mantissa has the normalised form `1.xxxxx`, and we don't need to store the 1, since we know it's already stored. For example, 0.011×2^5 is equivalent to 1.1×2^3, but we use the latter as it makes better use of available bits. We also have a number of special cases;

- the bit pattern with all `0`s represents zero
- if the `exp` is all `1`s, and the `frac` is all `0`s, it represents ∞, which can be signed
- if the `exp` is all `1`s, and the `frac` isn't all `0`s, it represents `NaN`

Since we can't use all `0`s, or all `1`s in our exponent field, because it's reserved as mentioned above, we need to encode it with a bias B, such that we have $E = \texttt{exp} - B$. The bias is $2^{k-1} - 1$, which is 127 in single precision, and 1023 in double precision. Hence `exp` is in the range [1, 254], when the actual value of E is in the range [-126, 127], and the same for double precision [1, 2046], vs [-1022, 1023]. This allows for negative values, which thus allows for very small values. It's important to note how the range of `exp` isn't from 0 to 255, as those are reserved. We have an implied leading bit of `1`, which means that we have a minimum when we have the bit pattern `000...0`, which is $M = 1.0$, and the bit pattern `111...1` represents $M = 2.0 - \epsilon$. This is an example working on `float f = 12345.0;`

Value:

$$\begin{aligned} 12345_{10} &= \texttt{11000000111001}_2 \\ &= \texttt{1.1000000111001}_2 \times 2^{13} \quad \text{(normalised)} \end{aligned}$$

Significand:

$$\begin{aligned} M &= \texttt{1.1000000111001}_2 \\ \texttt{frac} &= \texttt{10000001110010000000000}_2 \quad \text{(drop the leading bit, and pad with 0s)} \end{aligned}$$

Exponent:

$$\begin{aligned} E &= \texttt{exp} - B \\ \Leftrightarrow \texttt{exp} &= E + B \\ E &= 13 \\ B &= 127 \\ \texttt{exp} &= 140 \\ &= \texttt{10001100}_2 \end{aligned}$$

Hence, the number is represented by $\underbrace{\texttt{0}}_{\texttt{s}}\ \underbrace{\texttt{10001100}}_{\texttt{exp}}\underbrace{\texttt{10000001110010000000000}}_{\texttt{frac}}$

Floating Point Operations, and Rounding

It's important to remember that floating-point representation isn't exact. Therefore we need to consider it as follows;

- `x` $+_f$ `y` — `Round(x + y)`
- `x` $*_f$ `y` — `Round(x * y)`

These operations require adjustment. For example, with addition, the exponents can be extremely different, and therefore they need to be aligned before addition is done. The general idea is to compute the exact result, then round the result to fit into the standard. Thre is possible overflow if the exponent is too large, or having to drop precision if it cannot fit into the mantissa.

If we were to repeatedly round, we can introduce a significant amount of error into the result. However, if we know the direction in which we are rounding in, we can introduce statistical bias to our results. The round-to-even method (which rounds to the nearest even number if it's at a mid-point (e.g 1.4 goes to 1, 1.5 goes to 2, -1.5 goes to -2 etc.)) can avoid this bias as we round up half the time, and down the other half. Therefore this is the default used in the IEEE standard.

If the exponent overflows, we can set the result to $\pm\infty$. Also, it's important to note that floating point operations aren't associative, or disassocitive, due to the rounding. For example, if we were to do `(3.14 + 1e10) - 1e10`, it would not be the same as `3.14 + (1e10 - 1e10)`, as the `3.14` is insignificant.

Floating Point in C

In C, we have `float`, and `double`, which are 32-bit, and 64-bit respectively. We should also avoid equality, as they can be unpredictable. The best approaches is to check the difference. Converting from a floating point to an integer in C, is simply truncating the fractional part of (and therefore rounds towards 0). It's not possible to overflow from an integer, to floating point representation, as floating points can represent much larger values (however precision may be lost).

To summarise; we have the following possible values;

value	`s`	`exp`	`frac`
zero	0	00000000	00000000000000000000000
normalized values	s	$[1, 2^k - 2]$	significand = 1.M
infinity	s	11111111	00000000000000000000000
NaN	s	11111111	non-zero
denormalized values	s	00000000	significand = 0.M

Architecture

I'm pretty sure this is covered in the first half of the module.

The time required to execute a program depends on the following facors;

- The program
- The compiler

  what set of assembler instructions it was translated into
- The ISA

  what set of instructions are available to the compiler
- The hardware

  how much time it takes per instruction

The ISA defines the system's state, such as the registers, memory, and the program counter. It also defines the instructions the CPU is able to execute, as well as the effects these instructions have on the system state. The ISA has to be designed with how memory is addressed, as well as the number of registers, and their widths. x86 processors currently dominate the server, desktop, and laptop markets. It's backwards compatible to 8086, which was introduced in 1978. IA32 is the traditional 32-bit x86 implementation, and the new standard of x86-64, which is 64-bit.

The states visible to the programmer are the Program Counter, holding the address of the next instruction (also referred to as "EIP" (extended instruction pointer) in IA32, and "RIP" in x86-64). The registers are also visible to the assembly code, as well as the condition codes which stores information about the most the recent arithmetic operation, this can be used for conditional branching. The memory is byte addressable array, which contains code, user data, and some OS data. Includes a stack to support procedures.