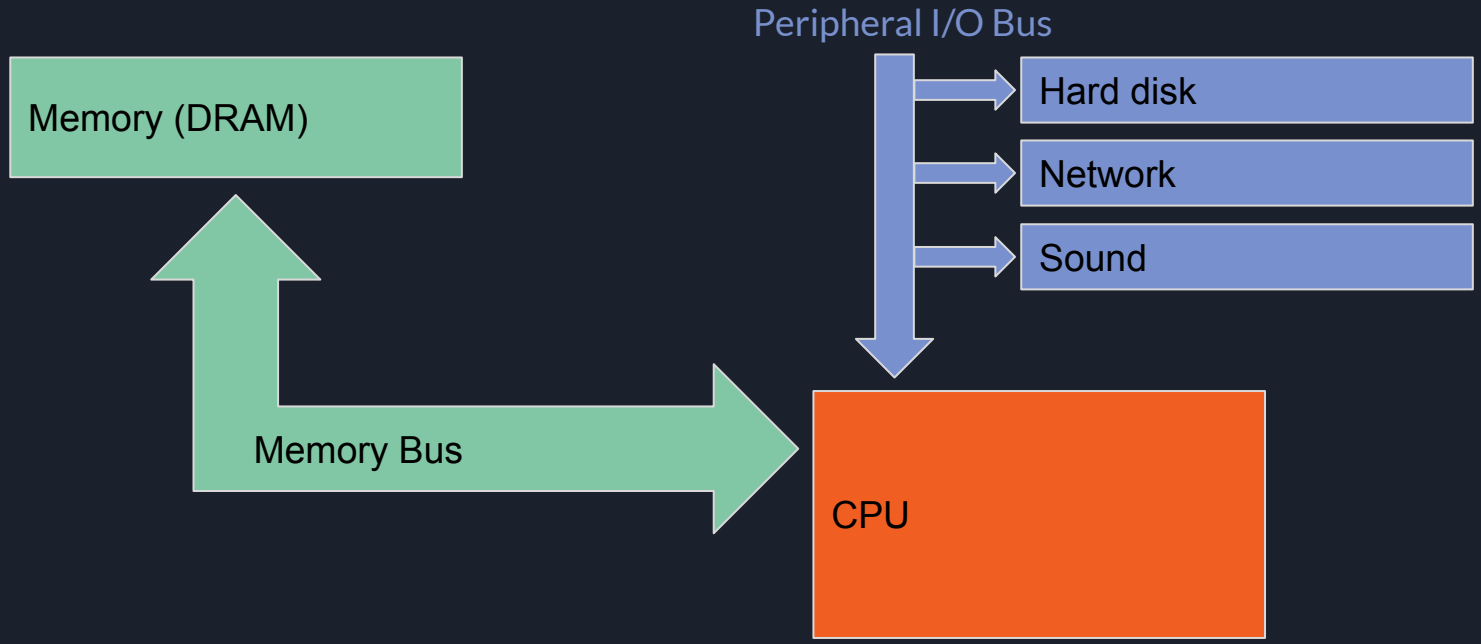
Session 1: Computer Architecture

Dr John Fawcett

The anatomy of a simple computer



What is Dynamic Random Access Memory?

- "*Memory*" describes the function of this hardware component: it holds information.
  - Data is not changed, updated, nor calculated.
  - It's just storage, so we don't forget all the numbers we're dealing with.
- "*Random Access*" means that we do not pay a performance penalty for accessing locations out of sequence.
  - Your rucksack is linear memory: you can access items easily in the reverse of the order in which you inserted them but finding items in a different order requires a lot of unpacking and repacking, which takes time.
  - Random access memory is the opposite: you can access locations in any order with the same access time.
- "*Dynamic*" refers to the circuit design. Dynamic circuits charge up capacitors to store 1s, and do not charge them to store 0s, then later test to see whether they are charged.
  - Needs refreshing every millisecond or so because the charge leaks away.

Memory Addresses

- Memory is like a very long corridor with numbered doors.
  - The number on each door is its *address*.
  - The rooms behind each door hold a single *byte*: a value from 0 to 255 (an 8-bit number)
- These are byte-addressed memories: each address refers to a byte
  - Supercomputers store more information at each address so they can handle more data with the same number of addresses.
- We usually need to store 32-bit and 64-bit data so we draw memory in rows of 4 bytes (32-bits). Each row begins on the next multiple of 4.
- We start numbering from zero (zero-based indexing)
- This example is storing the numbers 0 to 15 in the first 16 memory locations.

	+0	**+1**	**+2**	**+3**
0	0	1	2	3
4	4	5	6	7
8	8	9	10	11
12	12	13	14	15
...				

How much memory do you want?

- The more memory we have, the bigger the numbers we need to address it.
- To start this course, we are going to use a 32-bit memory address.
- We need to work out how much memory can be used if the addresses are limited to 32 bits.
  - The first memory location is zero:
    00000000 00000000 00000000 00000000
  - We cannot address anything after:
    11111111 11111111 11111111 11111111
  - So the number of memory locations is 2^{32}.
  - If each location holds 1 byte, then the capacity of our memory is 2^{32} bytes = 4,294,967,296 bytes = 4 GiB

	+0	+1	+2	+3
0	.	.	.	.
4	.	.	.	.
8	.	.	.	.
…	…	…	…	…
2^{32}	.	.	.	.

Accessing DRAM

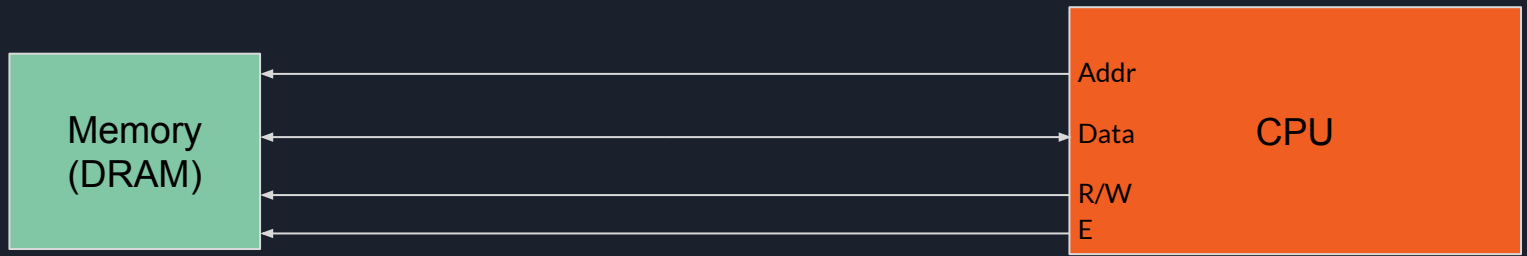
- You need to tell the DRAM whether you want to read from it or write into it.
- You need to provide the address
- If you are writing into it, you need to provide the data to be stored.

- Because we want to work with 32-bit values, all our addresses will be multiples of 4.
- In binary, all multiples of 4 have 00 in the least significant bits.
- That means our addresses only need 30 bits.

The Memory Bus

The memory bus is a collection of wires:

- 30 wires for the address (from CPU to Memory)
- 32 wires for the data (bidirectional)
- 1 wire to indicate whether to read/write (from CPU to Memory)
- 1 wire to enable the memory (from CPU to Memory), so it knows when to listen



CPU: Central Processing Unit

The CPU executes instructions.

Instructions are encoded as 32-bit values and are stored in memory.

To execute an instruction we need to…

- Know where in memory the next instruction is stored
- We need to load it from that memory address
- We need to decode it and execute it
- We need to update our “next instruction” address.

Let’s look at how this works in a modern CPU.

Instruction Pointer / Program Counter

The instructions pointer (IP), also known as the Program Counter (PC), is a 30-bit value stored inside the CPU using Static RAM (SRAM) technology.

- SRAM is *very* fast but uses a lot more energy per bit of data stored so we would not want to build our main memory with SRAM.
- We use SRAM where we need exceptionally fast access to small amounts of data.

The CPU is an example of clocked logic: everything it does is governed by a clock. The circuitry only does something when the clock 'ticks'. Modern CPU clocks tick with a frequency of 1,000,000,000 'ticks' per second to 4,000,000,000 'ticks' per second – ie. 1 to 4 GHz.

CPU Registers

As well as the IP/PC, a modern CPU contains ‘registers’, which hold the temporary data that the program is manipulating.

Most CPUs have one set of integer registers and a second set of floating point registers. We are going to use a CPU that has 16 registers of each kind.

The registers are also built using SRAM.

We will look at “load/store architectures”, where the instructions in the program tell the CPU to…

- load data from memory into registers so computation can be performed; then
- store (write) the resulting data back to memory.

Encoding instructions

We need to encode our instructions so we can store them in memory, as bytes.

- Instructions have an *opcode* that specifies what we want the CPU to do
  E.g. ADD, SUB, …
- Let's give ourselves 7 bits for the opcode (a 128 instruction CPU).
- Instructions have *operands*, which tell the CPU what to ADD together (for example), and where to put the result.
- Addressing modes tell us what format an operand takes
  - A register number; or
  - An *immediate* (a value encoded in an instruction), e.g. to make it possible to say R3 = R4 + 1

An instruction encoding

We have 16 registers so we need 4 bits every time we need to specify a register.

ALU instructions (such as ADD) need two inputs and one output register, called the destination. When both inputs are registers, we can use this:

7 bits	1 bit	4 bits	4 bits	4 bits	12 bits
Opcode	0	Destination register	Input register 1	Input register 2	unused

...and when the second operand is an immediate, we can use this format:

7 bits	1 bit	4 bits	4 bits	16 bits
Opcode	1	Destination register	Input register 1	Input 2 (immediate)

MSB LSB

Opcodes

We can define our opcodes in any way we like. The operations that we make available form the *instruction set architecture (ISA)* for our CPU.

0	HALT	8	AND
1	ADD	9	OR
2	SUB	10	NOT
3	MUL	11	…
4	DIV		
5	SHL		
6	ASR		
7	LSR		

Accessing Memory

To add a new feature to our CPU, we need to define new opcodes in the instruction set architecture. We need to LD (load from memory) and ST (store to memory).

0	HALT	8	AND
1	ADD	9	OR
2	SUB	10	NOT
3	MUL	11	LD
4	DIV	12	ST
5	SHL	13	…
6	ASR		
7	LSR		

Accessing Memory

In this case, we also need a new instruction format. We do not need a new 1-bit flag because the opcode (LD or ST) already tells us that the instruction will be encoded in this format.

To say “LD into Register <n> from Address <x>” we say “LD n x”: immediate addressing.

7 bits	1 bit	4 bits	20 bits
Opcode	0	Dest/Src register	Memory address

We can give ourselves *immediate displacement*: “LD into Reg <n> from Addr [Reg<m>+<x>]”

7 bits	1 bit	4 bits	4 bits	16 bits
Opcode	1	Dest/Src register	Base register	Offset (immediate)

Assembly code and machine code [1/4]

Machine code is the encoded format: a long list of numbers. This is what is stored in memory.

Assembly code is easier to read but describes a program in the same way.

Machine code	**Assembly code**	**Means**
2,18,48,0	ADD R1 R2 R3	R1 = R2 + R3

0	0	0	0	0	0	1	0	0	0	0	1	0	0	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
Opcode = 1							0	Dest Reg				Input 1				Input 2				Unused											
ADD								R1				R2				R3															

Assembly code and machine code [2/4]

In assembly code, we use “#” to provide operand 2 as an immediate.

Machine code	**Assembly code**	**Means**
7,82,0,7	MUL R5 R2 #7	R5 = R2 * 7

0	0	0	0	0	1	1	1	0	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1
Opcode = 3							1	Dest Reg				Input 1				Input 2															
MUL								R5				R2				7															

Assembly code and machine code [3/4]

To load from memory address 4096 into register 10, we can say this:

Machine code	**Assembly code**	**Means**
22,160,16,0	LD R10 #4096	R10 = mem[4096]

0	0	0	1	0	1	1	0	1	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
Opcode = 11							0	Dest Reg				Memory Address																			
LD								R10				4096																			

Assembly code and machine code [4/4]

If we have an array of numbers, a, in memory and it begins at the address held in R6, then we can implement a[3] = R12 using a ST instruction:

Machine code	**Assembly code**	**Means**
25,198,0,12	ST R12 [R6, #12]	mem[R6+12] = R12

0	0	0	1	1	0	0	1	1	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0
Opcode = 12							1	Src Reg				Base				Memory Address															
ST								R12				R6				12															

Our first program!

SUB R1 R1 R1	R1 = R1-R1	Set R1=0, regardless of the initial value
ADD R1 R1 #1	R1 = R1 + 1	Remember that “#” means “immediate”
ST R1 #4096	mem[4096]=R1	
ADD R1 R1 #1	R1 = R1 + 1	
ST R1 #4100	mem[4100]=R1	
ADD R1 R1 #1	R1 = R1 + 1	
ST R1 #4104	mem[4104]=R1	
ADD R1 R1 #1	R1 = R1 + 1	
ST R1 #4108	mem[4108]=R1	
ADD R1 R1 #1	R1 = R1 + 1	
ST R1 #4112	mem[4112]=R1	We have written 1,2,3,4,5 into memory.

Quiz questions

What value is held in R2 after these two programs (left and right)?

```
SUB R1 R1 R1
ADD R1 R1 #1
ST R1 #4096
ADD R1 R1 #1
ST R1 #4097
ADD R1 R1 #1
ST R1 #4098
ADD R1 R1 #1
ST R1 #4099
LD R2 #4096
```

```
SUB R1 R1 R1
ADD R1 R1 #1
ST R1 #4096
ADD R1 R1 #1
ST R1 #4095
ADD R1 R1 #1
ST R1 #4094
ADD R1 R1 #1
ST R1 #4093
LD R2 #4096
```

Assume the least significant byte of a value is held in the lowest memory address. This is known as *little endian format*.

Answers

67,305,985

0

Understanding the left-hand program

SUB R1 R1 R1	R1=0
ADD R1 R1 #1	R1=1
ST R1 #4096	mem[4096]=1, mem[4097]=0, mem[4098]=0, mem[4099]=0
ADD R1 R1 #1	R1=2
ST R1 #4097	mem[4097]=2, mem[4098]=0, mem[4099]=0, mem[4100]=0
ADD R1 R1 #1	R1=3
ST R1 #4098	mem[4098]=3, mem[4099]=0, mem[4100]=0, mem[4101]=0
ADD R1 R1 #1	R1=4
ST R1 #4099	mem[4099]=4, mem[4100]=0, mem[4101]=0, mem[4102]=0
LD R2 #4096	mem[4096..4099]=(1,2,3,4)=$4*2^{24} + 3*2^{16} + 2*2^{8} + 1$

Understanding the right-hand program

SUB R1 R1 R1	R1=0
ADD R1 R1 #1	R1=1
ST R1 #4096	mem[4096]=1, mem[4097]=0, mem[4098]=0, mem[4099]=0
ADD R1 R1 #1	R1=2
ST R1 #4095	mem[4095]=2, mem[4096]=0, mem[4097]=0, mem[4098]=0
ADD R1 R1 #1	R1=3
ST R1 #4094	mem[4094]=3, mem[4095]=0, mem[4096]=0, mem[4097]=0
ADD R1 R1 #1	R1=4
ST R1 #4093	mem[4093]=3, mem[4094]=0, mem[4095]=0, mem[4096]=0
LD R2 #4096	mem[4096..4099]=(0,0,0,0)=0

Loops

So far, we have no way to create a loop. We need two new instructions:

1. Unconditional jump: always go to memory address X for the next instruction
2. Conditional branches: if <some test> then goto X else continue

Let’s add these to our instruction set architecture…

Unconditional jumps

0	HALT	8	AND
1	ADD	9	OR
2	SUB	10	NOT
3	MUL	11	LD
4	DIV	12	ST
5	SHL	13	JMP
6	ASR	14	…
7	LSR		

Encoding unconditional jumps

The target address might be a number of instructions (presented as an immediate) forwards or backwards in the program, or held in a register (with displacement).

Program counter relative jumps can be encoded like this:

7 bits	1 bit	24 bits
Opcode	0	2’s complement signed number of instructions relative to the JMP

...and when the target is register indirect with immediate displacement, we can use this format:

7 bits	1 bit	4 bits	20 bits
Opcode	1	Target base (Reg)	Offset (signed immediate)

MSB LSB

2’s complement notation

A binary representation of integers that can represent negative values as well as zero and positive values. Here’s a 10-bit example. The bit positions represent these contributions to the total. Note that the leftmost (most significant) is *negative*!

-512	256	128	64	32	16	8	4	2	1
0	0	0	1	0	0	1	0	0	0

= 64 + 8 = 72 (nothing new here!)

1	0	0	1	0	0	1	0	0	0

= -512 + 64 + 8 = -440

Clever trick

Note that we store the number *of instructions* to jump forwards or backwards, not the number of bytes! This is because all instructions are 4 bytes long so to jump by J instructions we update the program counter (PC) like this:

PC = PC + 4*J

This saves two bits and means J can represent jumps covering 4 times as many instructions as it would otherwise have been able to do. This is important when we have large programs with many thousands or millions of machine code instructions!

Conditional branches

To implement conditional branches, we need to have something we can examine and we need to specify what we want to check for.

When we run any instruction that updates a register, we can work out whether the destination register holds zero (all 32 bits are 0) or a negative number (most significant bit is 1 in 2's complement). Both are easy and fast checks to implement in the hardware of a CPU.

Example:

```
SUB R2 x y      R2 = x-y
BNE t           If the subtraction left R2 != 0 then goto t
```

Conditional branches

0	HALT	8	AND	16	BGT
1	ADD	9	OR	17	BGE
2	SUB	10	NOT	18	BLT
3	MUL	11	LD	19	BLE
4	DIV	12	ST	20	...
5	SHL	13	JMP		
6	ASR	14	BNE		
7	LSR	15	BEQ		

Branch if NotEqual (to zero), Branch if Equal (to zero)
Branch if GreatherThan, Branch if GreaterOrEqual
Branch if LessThan, Branch if LessOrEqual

Conditional branches

Branch if NotEqual (to zero)	NOT(Z)
Branch if Equal (to zero)	Z
Branch if GreatherThan	NOT(Z) AND NOT(N)
Branch if GreaterOrEqual	NOT(N)
Branch if LessThan	N
Branch if LessOrEqual	N OR Z

All six options are simple logic to implement in hardware!

Encoding conditional branches

The target address might be a number of instructions (presented as an immediate) forwards or backwards in the program, or held in a register (with displacement).

Program counter relative jumps can be encoded like this:

7 bits	1 bit	24 bits
Opcode	0	2’s complement signed number of instructions relative to the Bxx

…and when the target is register indirect with immediate displacement, we can use this format:

7 bits	1 bit	4 bits	20 bits
Opcode	1	Target base (Reg)	Offset (signed immediate)

MSB LSB

Our second program!

```
SUB R1 R1 R1      R1 = R1-R1      Set R1=0, regardless of the initial value
SUB R2 R2 R2      R2 = R2-R2      Set R2=0, regardless of the initial value
SUB R3 R3 R3      R3 = R3-R3      Set R3=0, regardless of the initial value
ADD R2 R2 #5      R2 = 5          Initialise loop counter: 5 iterations to go
ADD R3 R3 #4096                   Initialise target address to write to
ADD R1 R1 #1                      Beginning of loop (add 1 to get the next value to store)
ST R1 [R3,#0]                     Store it to memory at the address in R3
ADD R3 R3 #4                      Advance R3 ready for the next iteration
SUB R2 R2 #1                      Decrease number of iterations remaining
BNE #-4                           If we are not finished, go around again
                                  This achieves the same as our first program!
```

Comparing the programs

FIRST PROGRAM

```
SUB R1 R1 R1
ADD R1 R1 #1
ST R1 #4096
ADD R1 R1 #1
ST R1 #4100
ADD R1 R1 #1
ST R1 #4104
ADD R1 R1 #1
ST R1 #4108
ADD R1 R1 #1
ST R1 #4112
```

11 instructions

SECOND PROGRAM

```
SUB R1 R1 R1
SUB R2 R2 R2
SUB R3 R3 R3
ADD R2 R2 #5
ADD R3 R3 #4096
ADD R1 R1 #1
ST R1 [R3,#0]
ADD R3 R3 #4
SUB R2 R2 #1
BNE #-4
```

10 instructions

The second program is shorter but does it run faster?

Naive performance comparison

FIRST PROGRAM

11 instructions take 11 cycles to execute

11 cycles

SECOND PROGRAM

SUB R1 R1 R1	1 cycle
SUB R2 R2 R2	1 cycle
SUB R3 R3 R3	1 cycle
ADD R2 R2 #5	1 cycle
ADD R3 R3 #4096	1 cycle
ADD R1 R1 #1	1 cycle, 5 times
ST R1 [R3,#0]	1 cycle, 5 times
ADD R3 R3 #4	1 cycle, 5 times
SUB R2 R2 #1	1 cycle, 5 times
BNE #-4	1 cycle, 5 times

30 cycles

This is called the dynamic instruction count. (Static 11 vs 10, dynamic 11 vs 30.)

Better performance comparison

Our naive counts overlooked several factors!

1. We have to load the instructions from memory before we can execute them
2. Instructions that access memory take a second cycle to execute
3. CPU operations on registers take 1 clock cycle
4. Operations involving memory take about 600 clock cycles – memory is slow!

Let’s count again…

Better performance comparison

FIRST PROGRAM	load	execute
SUB R1 R1 R1	600	1
ADD R1 R1 #1	600	1
ST R1 #4096	600	600
ADD R1 R1 #1	600	1
ST R1 #4100	600	600
ADD R1 R1 #1	600	1
ST R1 #4104	600	600
ADD R1 R1 #1	600	1
ST R1 #4108	600	600
ADD R1 R1 #1	600	1
ST R1 #4112	600	600
9606 cycles		

SECOND PROGRAM	load	execute
SUB R1 R1 R1	600	1
SUB R2 R2 R2	600	1
SUB R3 R3 R3	600	1
ADD R2 R2 #5	600	1
ADD R3 R3 #4096	600	1
ADD R1 R1 #1	600	1, 5 times
ST R1 [R3,#0]	600	600, 5 times
ADD R3 R3 #4	600	1, 5 times
SUB R2 R2 #1	600	1, 5 times
BNE #-4	600	1, 5 times
21025 cycles		

The second program is 10% shorter but 2.2x slower!

Improving performance

Modern CPUs use a lot of tricks to increase performance. Huge gains result from caching instructions.

An instruction cache sits between the CPU and memory. It takes time to search the cache, even if the cache does not contain what you're looking for. When the cache *hits* (does contain what you're looking for), the access time is so fast it can happen in the same clock cycle as executing the instruction – zero access time!

Let's see how an instruction cache affects our two programs.

Comparing performance with I-caching

FIRST PROGRAM	load	execute
SUB R1 R1 R1	600	1
ADD R1 R1 #1	600	1
ST R1 #4096	600	600
ADD R1 R1 #1	600	1
ST R1 #4100	600	600
ADD R1 R1 #1	600	1
ST R1 #4104	600	600
ADD R1 R1 #1	600	1
ST R1 #4108	600	600
ADD R1 R1 #1	600	1
ST R1 #4112	600	600
9606 cycles		

SECOND PROGRAM	load	execute
SUB R1 R1 R1	600	1
SUB R2 R2 R2	600	1
SUB R3 R3 R3	600	1
ADD R2 R2 #5	600	1
ADD R3 R3 #4096	600	1
ADD R1 R1 #1	600*	1, 5 times
ST R1 [R3,#0]	600*	600, 5 times
ADD R3 R3 #4	600*	1, 5 times
SUB R2 R2 #1	600*	1, 5 times
BNE #-4	600*	1, 5 times
* = miss on the first iteration then 4 hits		
9025 cycles		

The second program is 10% shorter and now 6% faster! The I-cache is awesome!

Now it’s time for you to have a go

Exercise sheet!