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, multiply 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 minimise 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. Implementing 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 $x^2 + y^2$);

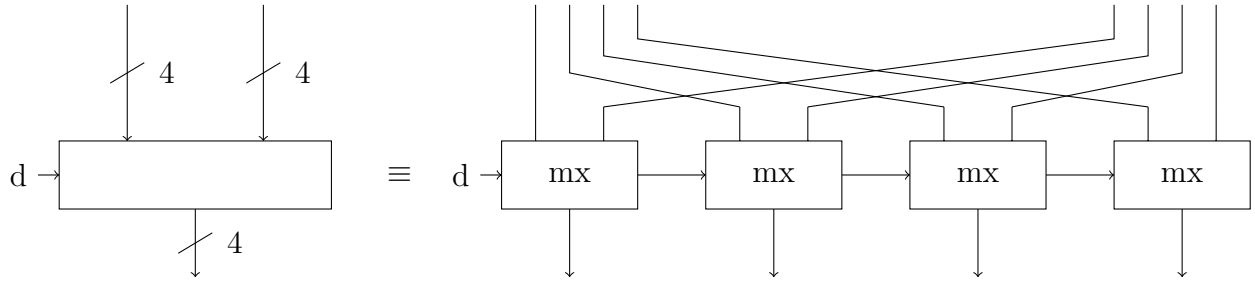
1. add instruction, accumulator, or load-store
2. add, and square instructions, accumulator, or load-store
3. custom sumsq instruction, with a dedicated circuit

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, where 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.

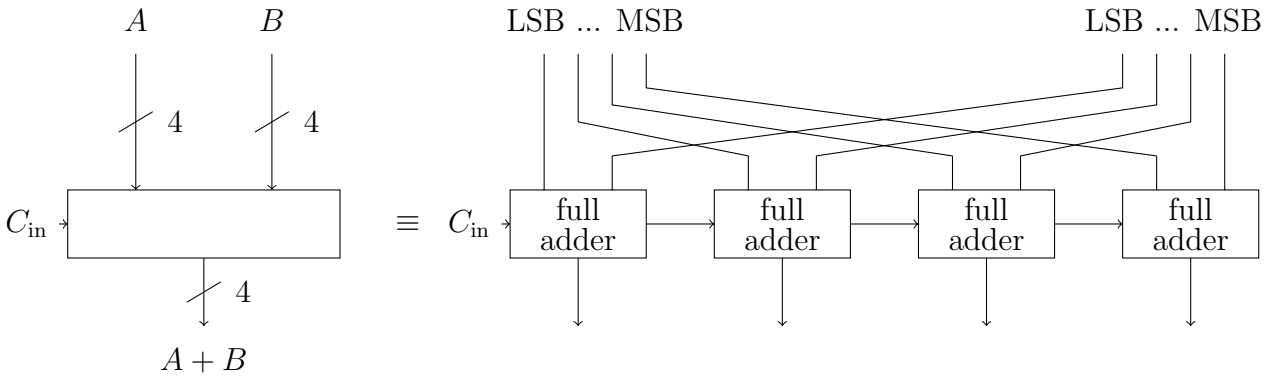
Lecture 4

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

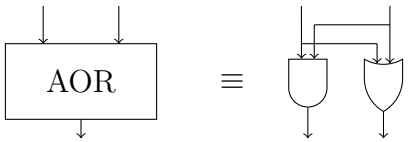
The layout of a MIPS ALU is similar to the basic one covered in **CO112**, as in, it has separate units for bit-wise AND, bit-wise OR, addition, etc. and also does the all the operations, then selects one based on the input. Similarly, it also uses the same slash notation to denote n lines being connected.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



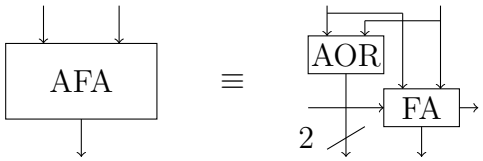
A similar diagram can also be used for the ripple carry adder, which joins n full adders, to create an n-bit ripple adder.



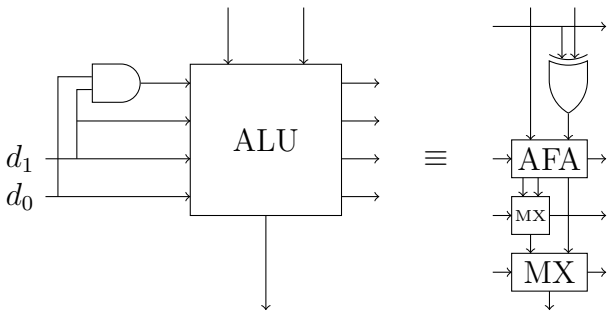
With this full adder, we are able to use this as as subtractor. For example, when working with $A - B$; take the ones complement of B (which is inverting B bitwise), and add it to A, and setting C_{in} to 1. Bitwise inversion is done with an XOR, when the other input is 1. Other important circuits found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

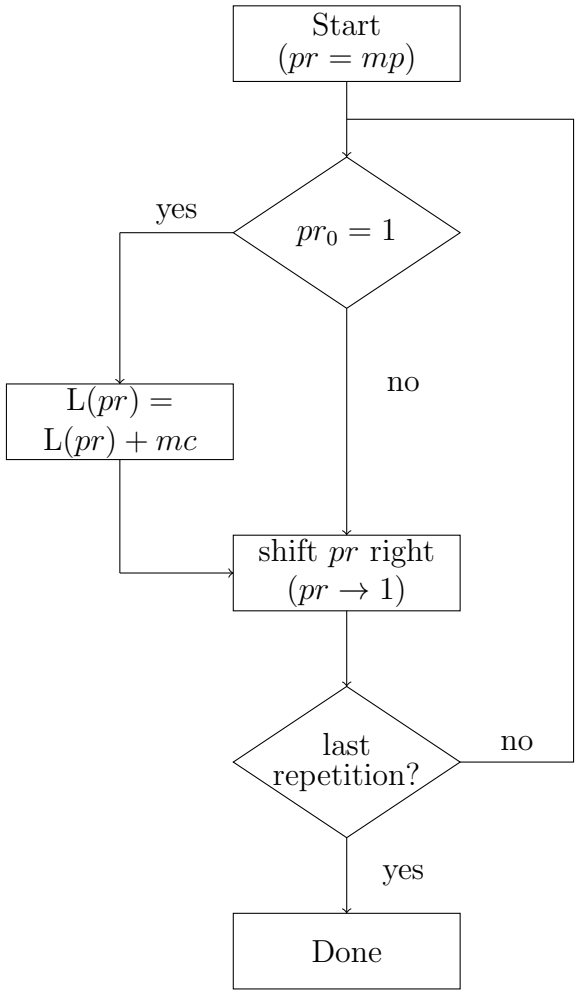
Consider the example $2 \cdot 11 = 22$, we have the multiplicand times the multiplier = product. However, to do this bitwise, we have to do the following (let $c \leftarrow n$ mean c shifted n bits to the left, and b_i mean the i^{th} bit of b);

				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 multiplicand shifted successively 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 L(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
	3: 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
	3: 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
	3: 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
	3: 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
	3: 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 multiplexers, which allow the right blocks to be connected, and finally the control unit unifies it by activating relevant parts of the combined datapath, based on the instruction.

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;

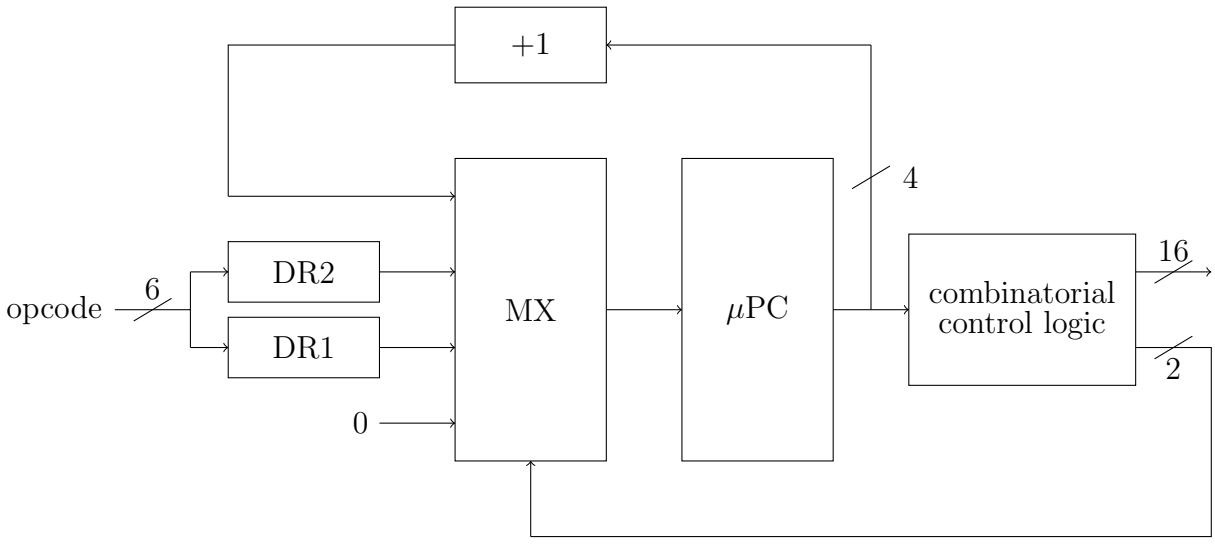
Step	R-type	memory-reference	branches	jumps
Instruction fetch	IR = M[PC] PC = PC + 4 S_0			
Instruction decode or register fetch	A = Reg[$IR_{25\text{-}21}$] B = Reg[$IR_{20\text{-}16}$] ALU_{out} = PC + (sign-extend($IR_{15\text{-}0}$) << 2) S_1			
Execution, address computation, branch or jump completion	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 achieve 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 individual 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 additional 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 indeterminate 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.

Addresses, 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 integral 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 implicitly 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 hardware, 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. There 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 distributive, 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
normalised values	`s`	$[1, 2^k - 2]$	significand = 1.M
infinity	`s`	11111111	00000000000000000000000
NaN	`s`	11111111	non-zero
denormalised 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 factors;

- 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.

Compiling, and Assembly

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.

Given two code files, `p1.c`, and `p2.c`, we compile it into (text) assembly, with `gcc -S`, resulting in `p1.s`, and `p2.s`. This is then assembled with `gcc`, or `as`, to get binary object programs `p1.o`, and `p2.o`. This is then linked with `gcc`, or `ld`, with static libraries to create the executable binary `p`. The compilation command is `gcc -O1 p1.c p2.c -o p` (the `-O1` is for optimisations). Here is an example of C code, and the IA32 assembly, obtained with `gcc -O1 -S code.c`;

```
1 /* code.c */
2 int sum (int x, int y) {
3   int t = x + y;
4   return t;
5 }
```

```
1 /* code.s */
2 sum:
3     pushl %ebp
4     movl %esp, %ebp
5     movl 12(%ebp), %eax
6     addl 8(%ebp), %eax
7     movl %ebp, %esp
8     popl %ebp
9     ret
```

As mentioned in the first part, there are three types of instructions, ones that perform arithmetic functions on register, or memory data, ones that transfer data between the main memory, and register, as well as ones that transfer control.

The basic data types are assembly, which are integer data types of 1, 2, or 4 bytes (in IA32), or 8 (in x86-64) bytes. These are data values, or they can be addresses (untyped pointers). There are also floating point data types, of 4, 8, or 10 bytes. Aggregate data types, like arrays and structs, don't exist in assembly. They are just sequentially allocated bytes in memory.

In particular, if we look at line 3 in `code.c`, that is analogous to line 6 in `code.s`. The C code adds two signed integers, whereas the assembly code is adding two 4-byte integers - it's the same instruction whether it's signed, or unsigned. In this case, we're using `%eax` as both the source register `x`, and the destination `t`. We can locate `y` in memory by getting `M[%ebp + 8]`, which is offsetting two memory locations away from the base pointer.

```
1 /* line 3 similar to */
2 x += y;
3 /* precisely */
4 int eax;
5 int *ebp;
6 eax += ebp[2];
```

Disassembling Object Code

We'll continue working on the `sum` example, which has the code, and can be disassembled into the following (both examples on the right side are equivalent, the first is with `objdump`, the second with `gdb`, the latter shows the offset from the initial entry);

```
0x401040 <sum>:
    0x55
    0x89
    0xe5
    0x8b
    0x45
    0x0c
    0x03
    0x45
    0x08
    0x89
    0xec
    0x5d
    0xc3
```

```
00401040 <_sum>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp, %ebp
   3:   8b 45 0c                mov    0xc(%ebp), %eax
   6:   03 45 08                add    0x8(%ebp), %eax
   9:   89 ec                   mov    %ebp, %esp
   b:   5d                      pop    %ebp
   c:   c3                      ret
```

```
00401040 <sum>:      push   %ebp
00401041 <sum+1>:    mov    %esp, %ebp
00401043 <sum+3>:    mov    0xc(%ebp), %eax
00401046 <sum+6>:    add    0x8(%ebp), %eax
00401049 <sum+9>:    mov    %ebp, %esp
0040104b <sum+11>:   pop    %ebp
0040104c <sum+12>:   ret
```

Anything that can be interpreted as executable code can be disassembled. Registers are at the heart of assembly program - they are a precious commodity in all architectures, but especially in x86. In x86, we have 6 general purpose registers (`%eax, %ecx, %edx, %ebx, %esi, %edi`), originally accumulate, counter, data, base, source index, and destination index (respectively) - but these are obsolete. There are also two special purpose registers (`%esp, %ebp`), which act as the stack pointer, and the base pointer (respectively). They are all 32-bit registers. However, for the purposes of backwards compatibility, we can drop the `e` from each register to access the right half (bits 15-0) (such that `%ax` refers to the least significant 16 bits of `%eax`). The first four general purpose registers listed can also be accessed even specifically, accessing bits 7-0 with `%al`, and accessing bits 15-8 with `%ah` - same for the other 3. With x86-64 registers, we once again have backwards compatibility, where you can refer to bits 31-0 of `%rax`, by using `%eax`. We also have another 8 additional registers, each accessible in the same way (through 8, 16, 32, or 64) bits. The additional registers `%r8` to `%r15` can be accessed with their least significant 32 bits with `%r8d` to `%r15d` respectively.

x86 Assembly

Move Operations

Remember that memory is indexed just like an array. Therefore we can load data from memory into a register as `%reg = M[addr]`, and store data into memory as `M[addr] = %reg`.

The general instruction for moving data is `movX src dest`, where `X` is either `b`, for a single byte, `w`, for a 2-byte word, or `l`, for a 4-byte long word.

The type of operands are as follows (in IA32);

- immediate : constant integer data

  this can only be used as a source, not destination

  encoded with either 1, 2, or 4 bytes

  prefixed with `$` (e.g. `$0x400`)

- register : one of 8 integer registers

  prefixed with `%`

  `mov %eax, %edx` copies the contents of register `%eax` into `%edx`

- memory : 4 consecutive bytes of memory at address given by register

  simplest example is using a register as an address, e.g `(%eax)` finds the data in the memory location specified by `%eax`

The possible combinations of `movl` are as follows. The hierarchy is in the form `src`, `dest`, instruction, C analogue;

`movl`	`Imm`	`Reg`	`movl $0x4, %eax`	`var_a = 0x4;`
		`Mem`	`movl $-147, (%eax)`	`*p_a = -147;`
	`Reg`	`Reg`	`movl %eax, %edx`	`var_d = var_a;`
		`Mem`	`movl %eax, (%edx)`	`*p_d = var_a;`
	`Mem`	`Reg`	`movl (%eax), %edx`	`var_d = *p_a;`

Note that you **cannot** do a memory-memory transfer in a single instruction. With memory access, there are multiple ways of accessing memory;

- indirect (R), means Mem[Reg[R]]
- displacement D(R), means Mem[Reg[R] + D]

  register R specifies a memory address, the constant displacement D is the offset from that address

Consider the following example, a basic function to swap variables between two memory locations;

```c
/* swap.c */
void swap(int *xp, int *yp) {
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

The following mappings are decided by the compiler;

register	value
%ecx	yp
%edx	xp
%eax	t1
%ebx	t0

The equivalent C actions are in comments below the assembly, to show how the program is executed step by step.

```
/* swap.s */
swap:
    /* setup */
    pushl %ebp
    movl  %esp, %ebp
    pushl %ebx

    /* body */
    movl  12(%ebp), %ecx
    /*    ecx = yp */
    movl  8(%ecx), %edx
    /*    edx = xp */
    movl  (%ecx), %eax
    /*    eax = *yp (t1) */
    movl  (%edx), %ebx
    /*    ebx = *xp (t0) */
    movl  %eax, (%edx)
    /*    *xp = eax */
    movl  %ebx, (%ecx)
    /*    *yp = ebx */

    /* restore stack, replace vals */
    movl  -4(%ebp), %ebx
    movl  %ebp, %esp
    popl  %ebp
    ret
```

x86-64 Assembly

In x86-64, we have an additional `q`, which is a quad word (8 bytes). Therefore we now have `movq`, in addition to `movl`, as well as the other instructions. You'll notice that the same program is much shorter

when optimised, since we're passing arguments in the registers, and this is much faster since we're not using the stack.

```c
/* swap.c */
void swap(long int *xp, long int *yp)
    {
  long int t0 = *xp;
  long int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```asm
/* swap.s */
swap:
    movq  (%rdi), %edx
    movq  (%rsi), %eax
    movq  %eax, (%rdi)
    movq  %ebx, (%rsi)
    retq
```

Addressing

The most general form of addressing memory in x86 is as as follows; `D(Rb, Ri, S)`, which computes `M[Reg[Rb] + S*Reg[Ri] + D]`, with the following variables;

- `D` — constant displacement value of 1, 2, or 4 bytes
- `Rb` — base register, which is one of the 8 / 16 integer registers
- `Ri` — index register, any register, except `%esp`, or `%rsp` (`%ebp` is also unlikely)
- `S` — scale, which is 1, 2, 4, or 8 (depends on array type size)

You can use the instruction `leal src, dest`, where `src` is an expression for the address mode. An example would be `leal (%edx, %ecx, 4), %eax`, which essentially does `%eax = %edx + 4*%ecx`

Some arithmetic operations we'll see often are the following. Note that there isn't a distinction between signed, and unsigned, as our arithmetic works regardless;

format	computation
`addl src, dest`	`dest = dest + src`
`subl src, dest`	`dest = dest - src`
`imull src, dest`	`dest = dest * src`
`sall src, dest`	`dest = dest << src`
`sarl src, dest`	`dest = dest >> src` (arithmetic, preserves sign)
`shrl src, dest`	`dest = dest >> src` (logical)
`xorl src, dest`	`dest = dest ^ src`
`andl src, dest`	`dest = dest & src`
`orl src, dest`	`dest = dest | src`
`incl dest`	`dest = dest + 1`
`decl dest`	`dest = dest - 1`
`negl dest`	`dest = -dest`
`notl dest`	`dest = ¬dest`

```c
/* arith.c */
int arith(int x, int y, int z) {
    int t1 = x + y;
    int t2 = z + t1;
    int t3 = x + 4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

```asm
/* arith.s */
arith:
    movl  8(%ebp), %eax
    movl  12(%ebp), %edx
    leal (%edx, %eax), %ecx
    leal (%edx, %edx, 2), %edx
    sall $4, %edx
    addl 16(%edp), %ecx
    leal 4(%edx, %eax), %eax
    imull %ecx, %eax
```

The compiler does optimisations here, and you can see that the order of the assembly code isn't necessarily the same as the order of instructions in the C code. A compiler could also optimise the program by setting constants, instead of recalculating it.

Conditionals, and Control Flow

In our high level languages, we have different conditional branches, such as `if`, `while`, `do`, `for`, etc, as well as unconditional branches such as break, and continue. In x86, these are all referred to as jumps, either conditional, or unconditional.

The jump instruction can depend on multiple things, but the ones we'll likely be using the most are unconditional jump (`jmp`), jump if equal (`je`), jump if not equal (`jne`), and some comparisons. However, with the signed ones, there's a number of variations. Other than our 8 registers, there is an additional register called the **instruction pointer** (`%eip`), as well as four single bit registers storing the condition of recent tests (`CF`, `ZF`, `SF`, and `OF`). These are implicitly set, which means they are set whenever instructions are carried out. For example, if we were to run `addl src, dest` $\leftrightarrow$ `t = a + b`;

- `CF` carry flag (unsigned)

  set if carry out from the most significant bit; hence unsigned overflow
- `ZF` zero flag

  set if `t == 0`
- `SF` sign flag (signed)

  set if `t < 0` (as signed)
- `OF` overflow flag (signed)

  set if two's complement (signed overflow)

  `(a > 0 && b > 0 && t < 0) || (a < 0 && b < 0 && t >= 0)`

However, this is **not** set by `lea`. In addition, we can also explicitly set these flags with `cmpl`, for example with `cmpl b, a`, we're doing `a - b`, without setting a destination;

- `CF` set if carry out from the most significant bit; hence unsigned overflow
- `ZF` set if `a == b`
- `SF` set if `a < b` (as signed)
- `OF` set if `(a > 0 && b < 0 && (a - b) < 0) || (a < 0 && b > 0 && (a - b) > 0)`

Additionally, we can set it with the `testl`, or `testq` instruction, for example, computing `testl b, a` is similar to computing `a & b` without setting a destination;

- `ZF` set if `a & b == 0`
- `SF` set if `a & b < 0`

There are also manual instructions that can be used to read the values of these condition bits into a register, which is used in the example below, where `setX` sets a single byte to `0` or `1` based on the combination of condition codes;

```c
/* gt.c */
int gt(int x, int y) {
  return x > y;
}
```

```
/* gt.s */
gt:
    movl   12(%ebp), %eax
    /* eax = y */
    cmpl   %eax, 8(%ebp)
    /* compare x and y (x - y) */
    setg   %al
    /* al = x > y */
    movzbl %al, %eax
    /* zeroes rest of al */
```

Conditional Branches

```c
/* absdiff.c */
int absdiff(int x, int y) {
  int result;
  if (x > y) {
    result = x - y;
  } else {
    result = y - x;
  }
  return result;
}
```

This example maps `x` to `%edx`, and `y` to `%eax`, hence we are doing `cmpl y x`, which is implicitly doing `x - y`, if `jle` is triggered, hence `x < y`, we jump to the point where we calculate `y - x`.

```
/* absdiff.s */
absdiff:
    pushl %ebp
    movl  %esp, %ebp
    movl  8(%ebp), %edx
    movl  12(%ebp), %eax
    cmpl  %eax, %edx
    jle   .L7
    subl  %eax, %edx
    movl  %edx, %eax
.L8:
    leave
    ret
.L7:
    subl  %edx, %eax
    jmp   .L8
```

However, in x86-64, there is a conditional move instruction. This is more efficient as it has a simpler control flow, and only moves from `src` to `dest`, if condition `C` holds (`cmovC src, dest`). This comes with a drawback, where there's more overhead since both branches need to be evaluated.

```c
/* absdiff.c */
int absdiff(int x, int y) {
  int result;
  if (x > y) {
    result = x - y;
  } else {
    result = y - x;
  }
  return result;
}
```

```
/* absdiff.s */
absdiff:
    movl   %edi, %eax /* eax = x */
    movl   %esi, %edx /* edx = y */
    subl   %esi, %eax /* eax = x-y */
    subl   %edi, %edx /* edx = y-x */
    cmpl   %esi, %edi /* (x - y) */
    cmovle %edx, %eax /* eax = y-x */
    /* ^ only if <= */
    ret
```

Some jumps are relative to the PC, and are therefore **relocatable**, but absolute branches aren't.

Loops

Something that was mentioned briefly in **CO141 - Reasoning About Programs**, was that most iterations are trivial to implement as a `while` loop. For example, we can look at the following code;

```c
/* loop.c */
void loop() {
  while (sum != 0) {
    <body>
  }
}
```

```
/* loop.s */
loop:
    cmpl $0, %eax
    je   loopDone
      <body>
    jmp  loop
loopDone:
```

In order to think about it in a way closer to machine code, we can use `goto`, which is something that is generally avoided in programming. This makes it a lot easier to implement, especially when we're dealing with more complex iterations, such as `for` loops, which will be discussed later. The use of a `goto` equivalent is fairly common in assembly, so understanding how code can be converted from our standard constructs to use it can be beneficial.

```
int fact_while(int x) {
  int result = 1;
  while (x > 1) {
    result *= x;
    x = x - 1;
  }
  return result;
}
```

```
int fact_while(int x) {
  int result = 1;
  goto middle;
loop:
  result *= x;
  x = x - 1;
middle:
  if (x > 1) {
    goto loop;
  }
  return result;
}
```

As previously mentioned, the conversion between a `for` loop, and a `while` loop is fairly trivial, and since we know how to go from a `while` loop to its `goto` equivalent, we can do the same for a `for` loop. Note that I only have the `start` there, since autogobble would remove the first two characters otherwise, and I'm really not bothered with redefining my environment; the flow is `for` → `while` → `goto`.

```
for (<init>; <test>; <update>) {
  <body>
}
```

```
<init>
while (<test>) {
  <body>
  <update>
}
```

```
start:
  <init>
  goto middle;
loop:
  <body>
  <update>
middle:
  if (<test>) {
    goto loop;
  }
done:
```

Switch Statement

Consider the following example of a `switch` statement, with many possible cases, and demonstrates many of the scenarios in a switch. For example, we have multiple labels, on 5, 6, fall through on 2, and missing cases. On the right is the corresponding jump table. Note that mini pages don't break on LaTeX, therefore I will do most of the writing up here (believe me, I'm not good with LaTeX). Below those two is the full assembly code (except for the jump table).

It's important to note that all of the cases which have a `break`, have the same ending two lines of assembly.

```
switch (x) {
  case 0:     <code>
              break;
  case 10:    <code>
              break;
  case 52000: <code>
              break;
  default:    <code>
              break;
}
```

It's also important for us to consider when it's actually viable to implement a jump table. In the example to the left, we would need a jump table which had 52001 entries, which is far too big. Jump tables should be reserved when we have a small range of dense cases to `switch`.

```
long switch_eg(unsigned long x, long y, long z) {
  long w = 1;
  switch(x) {
    case 1: /* .L56 */
      w = y*z;
      break;
    case 2: /* .L57 */
      w = y/z;
      /* fall through */
    case 3: /* .L58 */
      w += z;
      break;
    case 5:
    case 6: /* .L60 */
      w -= z;
      break;
    default: /* .L61 */
      w = 2;
      break;
  }
  return w;
}
```

```
.section .rodatra
    .align 4
.L62:
    .long  .L61 # x = 0
    .long  .L56 # x = 1
    .long  .L57 # x = 2
    .long  .L58 # x = 3
    .long  .L61 # x = 4
    .long  .L60 # x = 5
    .long  .L60 # x = 6
```

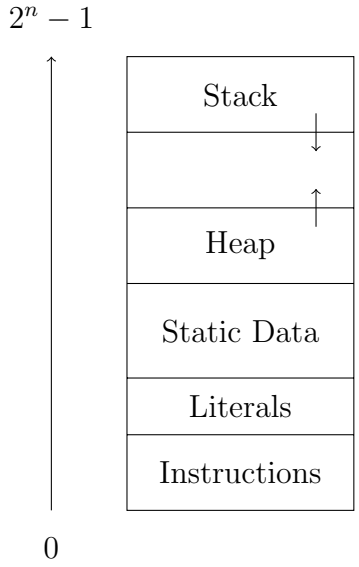
```
switch_eq:
    pushl %ebp            /* Setup */
    movl  %esp, %ebp      /* Setup */
    pushl &ebx            /* Setup */
    movl $1, %ebx         /* w = 1 */
    movl 8(%ebp), %edx    /* edx = x */
    movl 16(%ebp), %ecx   /* ecx = z */
    comp $6, %edx         /* x : 6 */
    ja   .L61             /* if > goto default */
    jmp  *.L62(, %edx, 4) /* goto table[x] */
.L61:                     /* default */
    movl $2, %eax         /* w = 2 */
    movl %ebx, %eax       /* return w */
    popl %ebx
    leave
    ret
.L56:                     /* case 1 */
    movl 12(%ebp), %ebx   /* w = y */
    imull %ecx, %ebx      /* w *= z */
    movl %ebx, %eax       /* return w */
    popl %ebx
    leave
    ret
.L57:                     /* case 2 */
    movl 12(%ebp), %eax   /* load y */
    ctld                  /* prepare for divide */
    idivl %ecx            /* y/x */
    movl %eax, %ebx       /* w = y/x */
/* fall through */
```

```
30 .L58:                    /* case 3 */
31     addl %ecx, %ebx      /* w += z */
32     movl %ebx, %eax      /* return w */
33     popl %ebx
34     leave
35     ret
36 .L60:                    /* cases 5&6 */
37     subl %ecx, %ebx      /* w -= z */
38     movl %ebx, %eax      /* return w */
39     popl %ebx
40     leave
41     ret
```

Procedures

Stacks

The general layout of our memory is as follows. It's important to note that it's possible for the stack to meet the heap (as the stack grows downwards, and the heap grows upwards) which would be problematic. This can be caused by going too deep into a procedure, or we're allocating too much memory, causing them to overwrite each other. It's also important to specify which regions are executable, for security purposes;



- stack — holds local variables, and procedure context
  writable, not executable, and automatically managed by the compiler
- heap (dynamic data) — holds variables allocated with `new`, or `malloc`
  writable, not executable, and managed by the programmer
- static data — `static` (including global) variables
  writable, not executable, and initialised at start
- literals — such as `"hello world"`, or `143`
  read-only, not executable, and initialised at start
- instructions — fairly self-explanatory
  read-only, executable, and initialised at start

In the IA32 call stack, we have the bottom of the stack at the highest memory address, and the top of the stack at the lowest address. The stack pointer, `%esp` holds the memory address of the top element, which is the lowest stack address. The stack generally grows downwards; therefore as we add things to the stack, the stack pointer gets lower, and as we take things off the stack - it gets higher.

When we run the instruction `pushl src`, we fetch the value from `src`, decrement `%esp` by 4 (4 bytes for a long), and store the value at the address given by `%esp`. On the other hand, when we run the corresponding `popl dest`, we do reverse - we load the value from the address pointed by `%esp`, and write that value to `dest`. `%esp` is then incremented by 4. We should note that the value isn't actually removed - we're just removing the reference to it.

Call Stacks

In general, a procedure call has a caller, and a callee. The caller first sets up arguments, and then calls the callee (which hands over control). The callee then creates its own local variables, runs some code, and then sets up the return value (to give back to the caller). It then cleans up its own local variables,

and then gives control back to the caller. The caller then finds the return value from the callee. The following rules have to be observed;

- callee must know where to find the arguments
- callee must know where to find the return address
- caller must know where to find the return value (hence know return address)
- caller and callee need to run on the CPU, as they use the same registers; this also means they need to save some registers if it may be used by the other

  this means that the caller has to save registers before setting up arguments, and then restore the registers after cleaning up the arguments, the same with the callee, but with local variables instead of arguments

This convention is called the **procedure call linkage**. Given a procedure call instruction `call label`, we push the return address on the stack (which is the instruction immediately after the call), then we jump to `label`. On the other hand, when we do a procedure return with `ret`, we pop the return the address from the stack, and jump to that address.

In this example, we put the address of the next instruction into `%eip`, push that value onto the stack, and then increment it by the arguments of `call` (we're using relative addressing). Note that `0x8048590 = 0x8048553 + 0x000063d`. Note that it's an arbitrary number of instructions between `0x8048590`, and `0x8048591`, we're just using the latter to denote the `ret` instruction.

```
1 804854e:    e8 3d 06 00 00    call    8048b90 <main>
2 8048553:    50                pushl   %eax
3 .......
4 8048591:    c3                ret
```

line	1	2	3	4
S: 0x110				
S: 0x10c				
S: 0x108	123	123	123	123
S: 0x104		0x8048553	0x8048553	0x8048553
%esp	0x108	0x104	0x104	0x108
%eip	0x804854e	0x8048590	0x8048591	8048553

By convention, we store the return value from a procedure in `%eax`. This is part of why it's important for the caller to save `%eax`, as this will be overwritten. If it doesn't fit in the 4 bytes, it's conventional just to return a pointer to the location where the return value is stored.

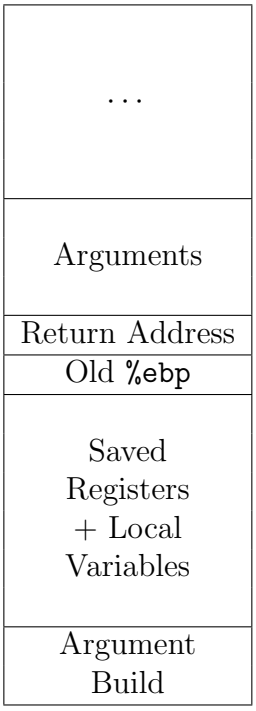
Stack-based Languages

Languages that support recursion are stack based languages, which requires code to be **re-entrant**, meaning that simultaneous instantiations of a single procedure is allowed. We need space to store the state of each instantiation, including arguments, local variables, and return pointers. The states are needed for a limited time, and the callee always returns before the caller does.

In order to allow this, we need to create stack frames, which store the data previously mentioned. The space is allocated when the procedure is entered, and then deallocated on return. Stack frames are allocated by moving `%ebp`, and `%esp`. There's a good graphical example on the corresponding video, which would take me too long to draw out.

IA32/Linux Stack Frame

We can construct the stack frame as such;



The initial space at the "bottom" of the stack (top of the diagram), is the memory used by the caller. The arguments are then put onto the stack, in the arguments section, and finally the return address, where the callee should return to, after completion. In the callee's frame, we also save the old pointer to the base of the caller's stack frame. Then there is more space for the callee's memory, and any arguments that might be used to call another function are put on the end.

Revisiting the previous example for `swap`, context is now provided for the offsets used.

```
1 swap:
2     pushl %ebp
3     movl  %esp, %ebp
4     pushl %ebx
5
6     movl 12(%ebp), %ecx
7     movl 8(%ebp), %edx
8     ...
```

offset from %ebp	resulting stack
	...
12	yp
8	xp
4	return addr
0 (%ebp)	old %ebp
-4 (%esp)	old &ebx

Registers, and Variables

Consider the following assembly code, where `func1` is the caller, and `func2` is the callee;

```
1 func1:
2     ...
3     movl $12345, %edx
4     call func2
5     addl %edx, %eax
6     ...
7     ret
```

```
1 func2:
2     ...
3     movl 8(%ebp), %edx
4     addl $54321, %edx
5     ...
6     ret
```

You'll notice that the register `&edx` is overwritten in `func2`, despite a value being already set for it in `func1`. The conventions here are **caller save**, and **callee save**, where the caller saves temporary values in its frame before calling, and the callee saves temporary values in its frame before using, respectively. `%eax, %edx, %ecx` are caller-save, and `%ebx, %esi, %edi` are callee-save. You'll note that `%eax` is also used to save the return value, which is why the caller must save it, since the value will be replaced by the callee on return. `%esp, %ebp` are special forms of callee-save, and are restored to the original value upon exit.

Actions that the caller does are highlighted in blue, and actions that the callee does are highlighted in violet;

1. set parameters in order (registers, and stack)
2. calls callee
3. save registers on stack
4. execute the body of method
5. copy any results to `%eax`/`%rax`
6. restore registers from stack
7. return to to caller
8. remove parameters from stack

9. use returned result

In summary, the IA32 procedures are a combination of instructions, and conventions. The latter prevents functions from disrupting each other's correct execution. The stack is used for the data structure because if `P` calls `Q`, then `Q` must return before `P` returns. In recursive cases, we can safely store values in local stack frames, and callee-saved registers. Arguments are in the top of the stack, and results are saved in `%eax`.

x86-64 Procedures, and Stacks

As we've doubled the number of general purpose registers, we can reduce the use of the stack. We can now store arguments, as well as temporary variables, in registers. If we run out however, we can default back to the stack. The usage conventions are as follows;

`%rax`	return value	`%r8`	argument #5
`%rbx`	callee saved	`%r9`	argument #6
`%rcx`	argument #4	`%r10`	caller saved
`%rdx`	argument #3	`%r11`	caller saved
`%rsi`	argument #2	`%r12`	callee saved
`%rdi`	argument #1	`%r13`	callee saved
`%rsp`	stack pointer	`%r14`	callee saved
`%rbp`	callee saved	`%r15`	callee saved

Previously, the different implementations of `swap` between IA32 versus x86-64 versions were compared. By avoiding the stack entirely (except for `ret`), we can make execution much faster as we don't have to fetch from main memory, and we have far less instructions.

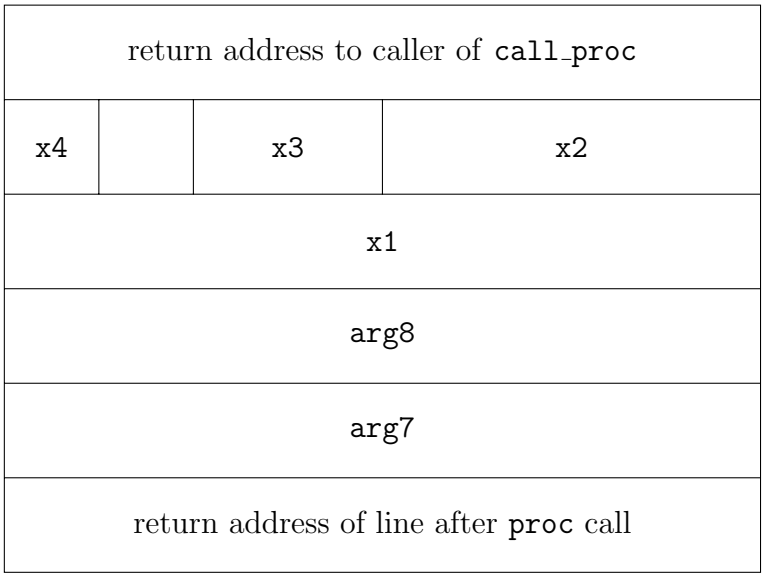
Local variables are also stored in the registers (if there is enough room), and the `callq` instruction stores 64-bit return addresses instead. We also have to decrement `%rsp` by 8, as we are now using 8 bytes. There isn't a frame pointer anymore, as all references are made relative to `%rsp`, and therefore both `%rbp` (and therefore `%ebp`) is now made available for use as a general purpose register. Functions are able to access memory up to 128 bytes beyond `%rsp`, which is referred to as the **red zone**.

Often, the x86-64 functions no longer need a stack frame, and the only use of the stack is just the return address for function calls. However, a function will need the stack frame when it has too many local variables for the registers, has more complex variables such as arrays, or structs. It will also need a stack frame if it uses the address-of operator (&) to compute a local variable's address, or if it calls another function which requires more than 6 arguments, or if it needs to save callee-save registers before modification. Once again, because minipages doesn't really allow me to split the content, I will have to do the majority of the writing up here (imagine the figure is between this paragraph, and the next).

It's important to note that the arguments have to be passed in the following order; `rdi, rsi, rdx, rcx, r8, r9`, and then finally into the stack. Note the penultimate 3 lines of the assembly code (lines 13-15) handle the 2 arguments that don't fit into the registers. The literal 4 is loaded into argument 7, which is where the stack pointer currently is, and the address is computed computed into `%rax`), and then moved from `%rax` into the correct location in the stack. Once we run the instruction `proc`, we push the return address onto the stack, and then run the remaining instructions. There's also the interesting instruction `movXYZ`, where `X` is either `z` for zeroes, or `s` for sign, `Y` is the original type, and `Z` is the target type, where it's extended from `X` to `Y`. We also add the literal 32 to the sign pointer to point back to reset the stack frame. The `cltq` instruction extends from a `long` to a `quad`.

In summary, x86-64 has a heavy use of registers, which is faster than utilising the stack in memory. If we use the stack, we allocate, and deallocate the entire frame at once. There's also more room for compiler optimisations, as we no longer have to do as much saving for register values.

```c
/* code.c */
long int call_proc() {
  long x1 = 1;
  int x2 = 2;
  short x3 = 3;
  char x4 = 4;
  proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4)
  return (x1 + x2) * (x3 - x4);
}
```

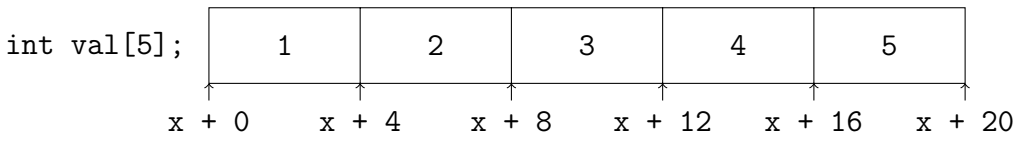



```
call_proc:
    subq   $32, %rsp
    movq   $1, 16(%rsp)
    movl   $2, 24(%rsp)
    movw   $3, 28(%rsp)
    movb   $4, 31(%rsp)
    movq   $1, %rdi
    leaq   16(%rsp), %rsi
    movl   $2, %edx
    leaq   24(%rsp), %rcx
    movl   $3, %r8d
    leaq   28(%rsp), %r9
    movl   $4, (%rsp)
    leaq   31(%rsp), %rax
    movq   %rax, 8(%rsp)
    call   proc
    movswl 28(%rsp), %eax
    movsbl 31(%rsp), %edx
    subl   %edx, %eax
    ctlq
    movslq 24(%rsp), %rdx
    addq   16(%rsp), %rdx
    imulq  %rdx, %rax
    addq   $32, %rsp
    ret
```

Arrays, and Structs

Arrays in C

Consider an array, called `A` of type `T`, and length `n`. Therefore, we'd require a **contiguously** allocated region of `n` · sizeof(`T`) bytes. The sizes are the same for primitive types, however if we were to have an array of pointers, the sizes would be different on IA32 machines versus x86-64 machines.



reference	type	value
`val[4]`	`int`	`5`
`val`	`int*`	`x`
`val + 1`	`int*`	`x + 4`
`&val[2]`	`int*`	`x + 8`
`val[5]`	`int`	`??` depends on what is at address `val+5`

Consider a slightly more complex example, where we iterate through an array, and its corresponding transformed version which uses pointers instead. We can eliminate the loop variable `i`, and instead have an ending pointer, increment the array pointer, and run checks in a `do-while` form;

```
int zd2int(int[] z) {
  int i;
  int zi = 0;
  for (i = 0; i < 5; i++) {
    zi = 10 * zi + z[i];
  }
  return zi;
}
```

```
int zd2int(int[] z) {
  int i;
  int *zend = z + 4;
  do {
    zi = 10 * zi + *z;
    z++;
  } while (z <= zend);
  return zi;
}
```

We're using the registers `%ecx, %eax, %ebx` to represent `z, zi, zend` respectively, as well as doing the computation `10*zi + *zi` as `*z + 2*(5*zi)`, and doing `z++` as an increment of 4.

```
start:                              /* not needed, autogobble breaks stuff */
    xorl %eax, %eax                 /* zi = 0 */
    leal 16(%ecx), %ebx             /* zend = z + 4 */
.L59:
    leal (%eax, %eax, 4), %edx      /* zi + 4*zi = 5*zi */
    movl (%ecx), %eax               /* *z */
    addl $4, %ecx                   /* z++ */
    leal (%eax, %edx, 2), %eax      /* zi = *z + 2*(5*zi) */
    cmpl %ebx, %ecx                 /* z : zend */
    jle  .L59                       /* goto loop if <= */
```

Multi-dimensional Arrays in C

In C, we use **row-major** ordering for all the elements, which essentially flattens the array. Consider the declaration of a multidimensional (nested) array; such that we have `T A[R][C]`, where `T` is the type, `A` is the name, and `R`, `C` being rows, and columns respectively.

$$\begin{bmatrix} \texttt{A[0][0]} & \cdots & \texttt{A[0][C-1]} \\ \vdots & & \vdots \\ \texttt{A[R-1][0]} & \cdots & \texttt{A[R-1][C-1]} \end{bmatrix}$$

In general, the size of this array would be $\texttt{R} \cdot \texttt{C} \cdot \text{sizeof}(\texttt{T})$. Therefore for some `T A[R][C]`; we'd have;

A [0] [0]	···	A [0] [C-1]	A [1] [0]	···	A [1] [C-1]	···	A [R-1] [0]	···	A [R-1] [C-1]

In general, the starting address of row `i` would be $\texttt{A} + \texttt{i} \cdot \texttt{C} \cdot \text{sizeof}(\texttt{T})$. In order to get the data at `A[i][j]`, we combine what we had previously (the starting index of the row), and combine it with the address equation from earlier on, to get $\texttt{A} + \texttt{i} \cdot \texttt{C} \cdot \text{sizeof}(\texttt{T}) + \texttt{j} \cdot \text{sizeof}(\texttt{T}) = \texttt{A} + (\texttt{i} \cdot \texttt{C} + \texttt{j}) \cdot \text{sizeof}(\texttt{T})$. It's also important to remember that the code does **not** do bound checking, and you have no guarantee of what is in memory beyond those ranges.

Multi-level Arrays

With a multi-level array, we're storing pointers to the "rows" (still using arrays of arrays as an example). This way, we don't have to store arrays contiguously - it's also how Java represents multi-dimensional arrays. Consider the following example, where we're using a multi-level array (note that the previous representation was skipped, since it essentially just did access as a flat array);

```
/* arr is a multi-level array of
   int[] */
int get_digit(int index, int digit) {
  return arr[index][digit];
}
```

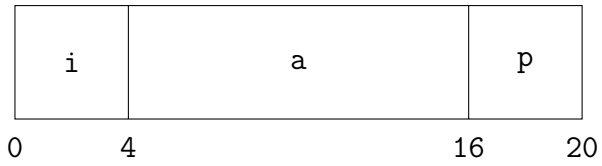
```
/* %ecx = index */
/* %eax = digit */
leal (, %ecx, 4), %edx
movl arr(%edx), %edx
movl (%edx, %eax, 4), %eax
```

It's important to note that we do two memory reads. The first one is to read the multi-level array to get the pointer to the row array, then do a second memory read to get the element within the row array. Obviously, this is slower since we have to do more memory accesses.

Structs

Let us consider a simple structure, as follows, with a corresponding memory layout of 20 bytes;

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```




```
struct rec r1;
r1.i = val;

struct rec *r = &r1;
(*r1).i = val;
r->i = val;
```

```
void set_i(struct rec *r,
           int val) {
  r->i = val;
}
```

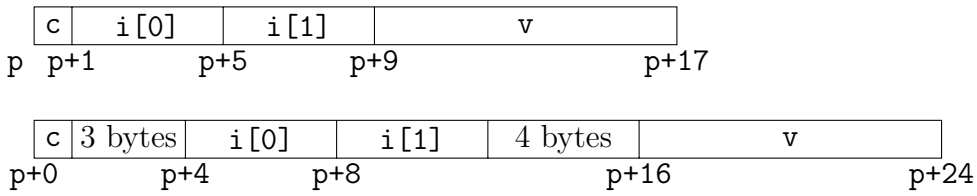
```
/* %eax = val */
/* %edx = r */
movl %eax, (%edx)
```

We often deal with pointers to structs, which is why we have the arrow shorthand in C. The second block of code is an example of how the arrow notation is used to set the value of an item when a pointer is given, and its equivalent assembly code.

Structs, and Alignment

Consider another struct, defined as;

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



For good performance, Intel recommends data to be aligned (x86-64 hardware will work regardless of data alignment). For data to be aligned, it means that a primitive object of size K bytes must have a memory address which is divisible by K. This is a requirement on some machines. The motivation for this is that we don't want to load, or store values which span word boundaries, as it is inefficient. This is handled by the compiler, but the specific alignment cases (on IA32) are as follows;

size (bytes)	examples	lowest address bits
1	`char`	no restriction
2	`short`	0_2
4	`int, float, char*`	00_2
8	`double`	000_2
16	`long double`	0000_2

Within the structure, each member has to satisfy it's alignment requirement, but the overall structure has alignment requirement K, where K is the largest alignment of any element in the structure. The initial address, and the structure length, must be a multiple of K. Note that in the example above, we wouldn't need the 4 empty bytes in IA32 Linux, but we would in IA32 Windows, or x86-64. Therefore sizeof(`S1`) is different in both of them, as in Linux it would be 20, whereas in Windows it would be 24. As a programmer, we can save space by putting the larger data types first.

Caches

The bus between the process, and main memory is much slower, at least compared to the computer. Small caches located close to the processor allows frequently used data to be stored for reuse. The general idea revolves around either a **hit**, or a **miss**. If the requested data is found in the cache, then it's the former, otherwise it is retrieved from main memory, and then stored in the cache.

Cache Locality

Caches work based on the idea of locality. Temporal locality suggests that recently referenced items are likely to be referenced again in the near future, and spatial locality suggests that items with nearby addresses tend to be referenced close together in time.

Caches are able to take advantage of temporal locality by storing recently used data, and takes advantage of spatial location by pulling entire blocks of data, after something is used. For example, consider this simple program that sums an array;

```
1 int i, sum = 0;
2 for (i = 0; i < n; i++) {
3   sum += a[i];
4 }
```

We have locality in the data, as we are accessing `sum` in each iteration (temporal), as well as accessing the array `a[]` in a stride-1 pattern. In addition, we also have locality in the instructions; we're cycling through the loop repeatedly, and also has spatial instructions as the instructions are typically stored sequentially. Consider this program, which doesn't take advantage of spatial locality;

```
1 int sum_array(int a[M][N]) {
2   int i, j, sum = 0;
3   for (j = 0; j < N; j++) {
4     for (i = 0; i < M; i++) {
5       sum += a[i][j];
6     }
7   }
8 }
```

Due to how C stores arrays, we're jumping all over memory, which means we cannot take advantage of spatial locality. However, if we were to iterate through `i` first, and then iterate through `j` inside, we'd essentially be summing a flat array, which takes advantage of spatial locality, like in the first example.

Memory Hierarchy

Consider that a cache hit time is 1 cycle, whereas a miss would be 100 cycles (Intel Haswell can process 512 bytes / cycle, whereas the bus has a bandwidth of 10 bytes / cycle, and a latency of 100 cycles). With a 99% hit rate, we have 1 cycle + $0.01 \cdot 100$ cycles = 2 cycles, compared to a 97% hit rate; 1 cycle + $0.03 \cdot 100$ cycles = 4 cycles. We need to discuss miss rate, instead of hit rate; the miss rate is the fraction of memory references not found in the cache ($\frac{\text{misses}}{\text{accesses}} = 1 - \text{hit rate}$. The typical number is between 3% to 10% for L1 cache. The hit time is the time to deliver a line in the cache to the processor, which is typically between 1 to 2 clock cycles for L1, and the penalty is the additional time required due to a miss (typically 50 - 200 cycles).

The general idea of memory hierarchy is that faster storage technologies tend to cost more per byte, and also have lower capacity. The gaps between memory technologies are widening, and well-written programs exhibit good locality, which means that slower memory can still give the illusion of speed. These properties complement each other, and suggest an approach for organising memory, and storage systems known as **memory hierarchy**. Each level serves as a cache for the level below it, which tends to be larger, slower, and also cheaper. Memory hierarchy is advantage of because programs tend to access data higher up in the hierarchy more often than the levels below it, which means that they can easily be slower, and not have a huge impact on performance. This creates the illusion that there is a large pool of memory that is fast.

The hierarchy goes; registers, on-chip L1 cache (SRAM), off-chip L2 (SRAM), main memory (DRAM), local secondary storage (local disks), and remote secondary storage (distributed file systems, servers). For example, in an *Intel Core i7*, each core has a separate L1 data cache, and L1 instruction cache, which are each 32KB, 8-way, and have a latency of 4 cycles, as well as a unified L2 cache, which is 256KB with an access latency of 11 cycles. Finally there is another L3 cache which is shared by all the cores, which is 8MB, and has an access latency of 30-40 cycles (probably outdated by now).

Cache Organisation

Consider a cache with a 8 blocks, in a 1-way cache, we'd have 8 sets, each with a block associated with it (direct mapped), a 2-way cache would have 4 sets, each with 2, and so on until 8-way with a single set, containing 8 blocks (fully associative). In a fully associative cache, data can go anywhere in the cache, which is expensive, but on the other hand a direct mapped cache can cause conflicts.

Consider a 2-way set associative cache, there are 2 places in a set, where a piece of data could go. Given an m-bit address, we have $m - k - n$ bits for the tag, k bits for the index, and n-bits for the block offset. This is on a block size of 2^n bytes. $k = \log_2(\text{number of sets})$. It's also important to remember that when you do a mod 2^b, you're taking the b least significant bits.

In a more realistic example, consider where the data in address `0x1833` would go. We have a block size of 16 bytes. In binary, `0x1833` is `00...00110011`. In an m-bit address, the offset is 4-bits, which is `0011`, in our case, and the index depends on the associativity we're using. Therefore, for a 1-way associative, we'd use index 3, for 2-way, we'd use index 3, and for 4-way, we'd use index 1 (we're taking the k least significant bits from 011). Typically, we kick out the least recently used block, as that preserves temporal locality.

If we say a set is 2^e-way associative, it means that there are 2^e lines (or blocks) in a set. Each block in the cache has metadata; a validity bit, a tag, and 2^b bytes of data per cache line (also known as the data block). The size of the cache is 2^{s+e+b} bytes. There are 2^s sets. The first thing that needs to be done when a request is made to memory; we need to locate the set in which the data may be located, and then check if any of the lines / blocks in the set has a matching tag. If the tag matches, we can then read from the offset (which is calculated as above).

There are three types of cache misses;

- cold (compulsory) miss

  occurs on the first access to a block

- conflict miss

  happens when multiple addresses map to the same set, and one is kicked out by another

- capacity miss

  the working set (active cache block) is larger than the cache

It's also important for us to consider writing to, and from the cache, not just reading. The main problem with writing is that the data between the hierarchies may disagree with each other. On a

write-hit, the possible options are to **write-through** it writes immediately to memory, or to **write-back**, it writes to the cache, and defers the write until the line is kicked out. This requires a **dirty bit** to indicate that the line is different from memory.

On the other hand, we need to consider what happens when we have a miss on a write. We can either do **write-allocate**, which loads the address into cache, and updates the line, which is beneficial if there will be more writes to the location, or to do **no-write-allocate**, which directly writes to memory instead. The typical cache is **write-back**, and **write-allocate**, although some machines are occasionally **write-through**, and **no-write-allocate**. The latter essentially bypasses the cache, as if it wasn't there at all.

Code Optimisation for Caches

In order to optimise our code, we should write code that takes advantage of locality, which means that we should aim to access data contiguously, and make sure that any repeated data access is close together in time (to avoid it being evicted from the cache). This can be achieved by proper algorithm choices, and loop transformations (consider the example earlier on).

The example where this is often applied to is matrix multiplication (square matrix in this point, but is trivial to generalise to other matrices);

```
1  c = (double *) calloc(sizeof(double), n*n);
2
3  void mmm (double *a, double *b, int n) {
4    int i, j, k;
5    for (i = 0; i < n; i++) {
6      for (j = 0; j < n; j++) {
7        for (k = 0; k < n; k++) {
8          c[i*n + j] += a[i*n + k]*b[k*n + j];
9        }
10     }
11   }
12 }
```

If we do a cache miss analysis, we need to make the following assumptions; matrix elements are doubles, the cache block is 64 bytes (which is 8 doubles), and the cache size is much smaller than $\mathtt{n}$.

On the first iteration, we have $\frac{\mathtt{n}}{8} + \mathtt{n} = \frac{9\mathtt{n}}{8}$ misses (given that we store 8 doubles, the first one will miss, but the rest will be saved by spatial locality). We also need to add $\mathtt{n}$, since it will miss on every single one of the second matrix. Now, in our cache, we have the last 8 elements of the first row of matrix X (we're thinking of this as $Z = XY$), and an 8×8 block at the bottom left corner of Y, so the first 8 doubles, of the last 8 rows. Given that this is done $\mathtt{n}^2$ times, the total miss is $\frac{9\mathtt{n}^3}{8}$. We can take advantage of the blocks, to create a blocked matrix multiplication algorithm;

```
1  c = (double *) calloc(sizeof(double), n*n);
2
3  void mmm (double *a, double *b, int n) {
4    int i, j, k;
5    for (i = 0; i < n; i+=B) {
6      for (j = 0; j < n; j+=B) {
7        for (k = 0; k < n; k+=B) {
8          for (i1 = i; i1 < i+B; i1++) {
9            for (j1 = j; j1 < j+B; j1++) {
10             for (k1 = k; k1 < k+B; k1++) {
11               c[i1*n + j1] += a[i1*n + k1]*b[k1*n + j1]
12             }
13           }
```

```
14            }
15          }
16        }
17      }
18  }
```

Now, when we run the same analysis, we're making the same assumptions about the block size, and the cache size, and we also assume that three blocks fit into the cache (such that $3B^2 < C$).

In the first iteration, we get $\frac{B^2}{8}$ misses for each block, and we have $\frac{\mathtt{n}}{B}$ blocks across on Y, therefore our miss is $\frac{2\mathtt{n}}{B} \cdot \frac{B^2}{8} = \frac{\mathtt{n}B}{4}$ misses. Our total misses will be $\frac{\mathtt{n}^3}{4B}$.

A programmer can optimise code for cache performance, and we need to consider blocking as a general technique. However, getting absolute optimum performance depends on the platform, as we need to consider how the cache is set up.

Lecture 16 / 17 - Caching

Apparently this is taught differently in our course. Oops. Some of the content here will probably have been covered above.

The values we use here, are as follows (note that `address` means the actual binary representation of the address);

- `address` size — A

  fairly self explanatory, but we use it to calculate `T`

- block size — K

  unit of transfer between the memory, and cache, and is a power of 2; consists of adjacent bytes for spatial locality

- offset field — lowest $\log_2(K) = O$ bits of `address`

  tells you which byte within a block, calculated by doing (`address %` K)

- cache size — C, or $\frac{C}{K}$

  given in bytes, or number of blocks, respectively

- index — next lowest $\log_2(\frac{C}{K}) = I$ bits of `address`

  a simple hash algorithm to determine which block to place the data in - the size will depend on the hash algorithm

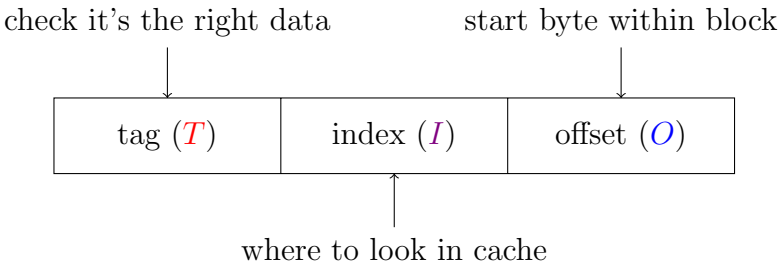
- tag — remaining $A - I - O = T$ bits of `address`

  for example, both the data in addresses `110100`, and `100101` would be placed in the block indexed `01` (with `00`, and `01` as the offset, respectively), therefore we'd tag the block with the remaining bits (`11`, and `10`) to differentiate them

- associativity — N

  will be explained further below

Therefore, the address is then broken down into;

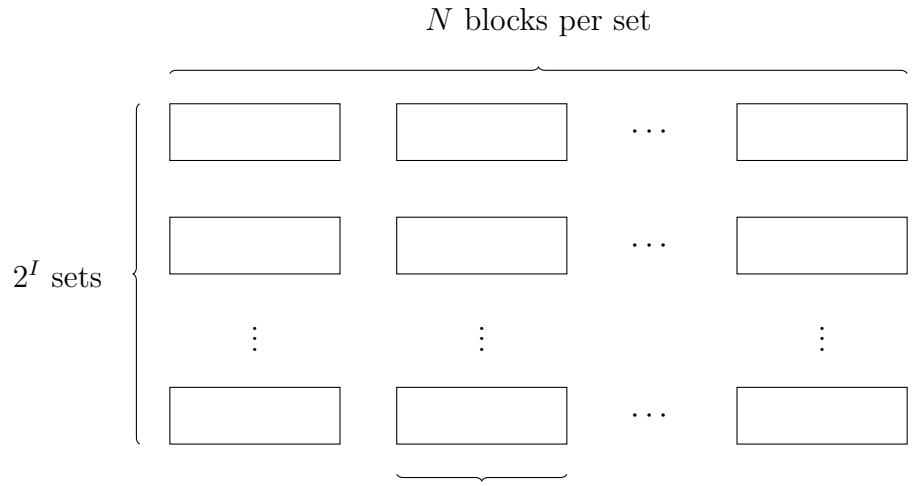


However, this naive approach can be problematic. Consider the example, where we have a hash collision - if we were to call `110100`, and `100101` repeatedly, we wouldn't have utilised the cache well, since those would constantly be evicting each other, thus removing any temporal locality.

We can generalise our previous approach by having the indices represent sets, instead of individual blocks, such an N-associative set will have N blocks per set. Our original approach was a direct mapping, which counts as 1-associative, as opposed to a single set, which would be fully associative. We refer to the block address as the memory address, without the offset (?). This means the amount of bits allocated to the index (I) changes, and therefore the bits for the tag (T) will also change;

$$\log_2(\tfrac{\frac{C}{K}}{N}) = I, \text{ so when } N = 1,\ I = \log_2(\tfrac{C}{K}), \text{ from before, and when } N = \tfrac{C}{K},\ I = 0$$

When we want to read data, and encounter a miss, any empty block in the corresponding set can be used to store data. However, in the case there aren't any free blocks, we tend to replace the least recently used (or not most recently used) block, to maintain temporal locality.



Therefore, the cache size is $C = S \times N \times K$, without including the validity bit, nor the tag.

On a cache read, we first locate the set, and check if any of the blocks / lines in the set containing a matching tag. If there is a match, and the valid tag is set, then we can locate the data starting at the offset.

The rest of the content is covered in the University of Washington series.