Computer Systems Organization

Druhan Rajiv Shah

Spring - 2023

Contents

Course Structure

The first half (upto, but not including Assembly code) will be taught by Suryajith Chillara, while the second half will be taught by Girish Verma.

Textbooks

- Computer Systems: A Programmer's Perspective - Bryant & O'Hallon
- Computer Systems Organization - Patterson & Henessey

Topics The ranges in the parentheses are sections in the first textbook.

- Computer Arithmetic (2.1 - 2.3)
- Assembly language (3.4 - 3.7)
- Processor architecture and design (4.1 - 4.5)
- Memory hierarchy (6.1 - 6.4)
- System calls: Intro to process control (8.2 - 8.4)

Grading

Assignments (Programming-based) - 20%

Deep Quizzes - 20%

Midsem Lab Exam - 15%

Endsem - 25%

Endsem Lab Exam - 20%

1 Introduction

How is this different? In previous courses we studied how computers worked in a sort of abstract way, without actually worrying about what goes on under the table. This 'under the table' stuff is exactly what we will be discussing in this course.

2 Computer System structure

Overall structure In a typical system, there are several functional sections, like the CPU, which contains the Program counter, register file and ALU; I/O Bridge, controllers, adapters and Main memory; each of which is interconnected through buses. These buses transfer fixed-side chunks of bytes (called words).

Now, all of the information is contained in bits (which, recall, are just states of low vs high voltage). The bandwidth of these buses is what determines the word length of the system. This would be more familiar as whether a system has a 32-bit or 64-bit architecture.

Note, the difference between a controller and an adapter is whether it is directly connected to the motherboard or not. So, mouse and keyboard ports are termed controllers, while a monitor or a display would need an adapter. Earlier, RAMs used to be connected using adapters as opposed to controllers.

CPU A CPU contains a PC, a register file and an ALU. Program counters are necessary for parallel processing of instructions. An ALU is self-explanatory and does most of the calculation here. Note that this is what CPUs looked like a good while ago. Nowadays it's a lot more complicated.

What CPUs do essentially, is they Load data from the RAM into the register; Store a copy form the register to the main memory; Operate by copying contents of a register to the ALU and load the output back into the register; and Jump by extracting a word from instrutions and overwrite into the PC. (Load, Store, Operate, Jump is gonna be a recurring theme in formalizations)

Dynamic RAM The main memory is organized into a linear array of bytes, each with its unique address (recall CPro). The Program Counter back in the CPU points to the curent instruction in the main memory.

Register file It's a small storage device that contains a collection of word-sized registrs, each with unique names.

Running an executable First, the system accepts the input of the command from the USB controllers, dumps it into the main memory and then processes it, loads the instructions from the disk into main memory, and finally runs each of the instructions in sequence.

Processor-Memory gap Moore's law says that the processing power doubles every constant amount of time (paraphrase :P).

RAM types The damage this causes can be mitigated by involving Cache memories and Registers, both of which are implemented using Static RAMs, which use flip-flops as opposed to capacitors and transistors in DRAM. DRAMs have a constant recharging of the capacitors, the rate of which is called the refresh rate of the DRAM. However, SRAMs use flip-flops which are significantly more stable. This makes SRAMs more complicated to design while making them significantly faster. This is also why SRAMs are used inside he CPU, while the main memory uses DRAM.

Note that both SRAMs and DRAMs are both volatile, meaning that the moment the system is shut down, the information stored in the RAM is destroyed.

More RAM types! There is a hierarchy of 7 types of RAMs, which are labelled from L0 to L6.

The faster, more efficient types of RAM also happen to be way more expensive than their cheaper counterparts and use more energy, which is why we need to optimize where we use each type of RAM and how much it would cost.

OS Cameo Operating Systems behave as a sort of middleman between the Application programs and the hardware of the system. They have two main purposes:

- To protect the hardware from misuse by runaway programs
- To provide a simple and uniform mechanism for programs to manipulate complicated and varied hardware devices (this is done through abstractions that are handled by device drivers)

The abstraction reaches comical levels when considering the implementation of a mouse and a keyboard as simply reading from a file associated with each (on Linux at least)! This is not too slow because these files are specially implemented to be a part of the main memory instead of disk, so reading and writing these files is not too slow.

The abstraction for running a program is a Process. This is convenient to handle concurrency of processes.

3 Computer Arithmetic

Numerical representation We usually use bit representations of numbers, which is convenient because storing information in bits is a lot more efficient because of the 'ON/HIGH' versus 'OFF/LOW' states. This was especially useful back in the days of punch card programming.

However, the representation is rather verbose. (In fact, base b is $\mathcal{O}\left(\frac{\log b}{\log a}\right)$ times as verbose as base a representation). On top of that, we are used to using base 10 due to the languages we speak and the writing systems we use. Conversion from base 10 to base 2 and vice versa is rather cumbersome.

To overcome this, we use the hexadecimal system instead, which has the added consequence of making conversion from binary to hexadecimal much more convenient. In fact, for a compiler, you can save some computation power and use a lookup table instead of calculating the conversion each time at the cost of a tiny amount of space.

Words Word size is a measure of the size of integer and pointer data.

Note that this means 32-bit computers have a hard limit on the size of virtual memory because there are only $2^{32} \approx 4 \times 10^9$ possible address indices.

Enter Jonathan Swift? The Lilliputian argument of Big-Endians vs Little-Endians is still strong in bit representations as well, because whether the first bit should be for the most or least significant bit respectively. *e.g.* Sun, IBM machines prefer the Big-Endian approach, while Intel machines prefer the Little-Endian approach.

The Big-Endian vs Little-Endian approach also applies to memory location indexing. Here the Little-Endian approach would require the least significant digits in the hexadecimal form of an address or integer to be stored and processed earlier in the memory.

Now this is dangerous because multiple machines which use different approaches in the same system could possibly lead to disaster if not spotted.

ASCII what you did there Strings can now be represented as strings of integers, each of which corresponds to each character. The generally accepted system of encoding the characters as integers is called the American Standard Code for Information Interchange (ASCII). However, there is also Unicode which is gaining more popularity over time.

3.1 Boolean operations

Basic operators The standard NOT, AND, OR and XOR are also used nowadays for bit operations in computers, even though the notation used is different from that used in Propositional Logic (because of the character set allowed by ASCII). These operations can be applied on bit vectors bit-by-bit to make 'bitwise' operators.

Additionally, $\oplus, \wedge, \neg$ can be thought of as $+, \times$, negation in the Boolean Ring.

As a bonus, bit vectors can be used to represent subsets (obviously), so the operators behave analogously to set operations in the Power Set Ring.

More representation Pure binary representation can only store nonnegative integers. If we were to store negative numbers as well, we would need alternative methods. One possibility is the signed n-bit notation, which assigns a sign bit at the cost of absolute range.

Another possibility, which is more often used, is the 2's complement encoding, which makes operations a lot simpler. Recall DSM.

$$\text{Signed } 2'\text{s complement} : \mathrm{B2T}_w(\vec{x}) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

$$\text{Unsigned representation} : \mathrm{B2U}_w(\vec{x}) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

Clearly, both of these functions are bijective over w-bit numbers.

Conversion between religions–wait no, I mean systems We can define $\mathrm{T2B}_w = \mathrm{B2T}_w^{-1}$ and $\mathrm{U2B}_w = \mathrm{B2U}_w^{-1}$ to be the inverses of the functions as above. Clearly these are more functions that covert from one representation to another. Now the conversion from signed to unsigned representations is going to cause some pain, but it is very much possible using these functions. Obtaining an intuition of these conversions is going to be necessary!

Words of a feather flock together Conversion of numbers from a higher-bit representation to lower is very messy and can lead to losses. However, converting from lower-bit numbers to higher ones is possible and very common. Suppose we convert a signed w-bit number to w'-bit number, where $w' > w$, then we must have that:

$$[x_{w-1}, x_{w-2}, \ldots, x_0] = [x_{w'-1}, x_{w'-2}, \ldots, x_0]$$
$$\Rightarrow x_i = x_{w-1} \quad \forall w' > i \geq w$$

This can be proved by considering positive and negative representations respectively. Understandably, the unsigned representation can just be converted by prepending zeroes in the beginning.

Also, it is relatively easily to see that truncating numbers is essentially taking the number modulo a power of 2. However, note that this gives us a strictly unsigned representation. Thus, if we wish to truncate a number x down to k bits, we will have to compute the value $\mathrm{U2T}_k\left(\mathrm{B2U}_k(x \mod 2^k)\right)$

Interestingly this can lead to some serious security flaws. See FreeBSD's `getpeername` vulnerability.

This operation's a success, doc! Operations on integers are not very hard to implement. However, the overflow situations need to be handled carefully. For unsigned integers, the overflow is simple as the number simply wraps around, essentially implementing a modulo operation after the addition. Signed representations are a little messier as they have to consider cases of positive and negative overflow.

We have

$$x +_w^u y = \begin{cases} x + y & x + y < 2^w \\ x + y - 2^w & 2^w \leq x + y \leq 2^{w+1} \end{cases}$$

For the signed two's complement case, things become spicier:

$$x +_w^t y = \mathrm{U2T}_w\left(\mathrm{T2U}_w(x) +_w^u \mathrm{T2U}_w(y)\right)$$

It can be proved that unsigned and signed multiplication can be computed identically as $x \cdot y \mod 2^k$ for k-bit integers.

Fraction representation Naturally, the mathematical definition of the base-n representation can account for fractional numbers. The powers simply extend in the negative direction.

Now, the challenge is to determine how to represent fractions with a fixed number of bits. If we use a fixed number of bits for the fractional part, then we cannot represent fractions of arbitrary precision. Instead, there will be some granularity, so to speak. We have to attain some middle ground, as far as possible.

Float like a butterfly... Now, the modern approach is the 'floating point', which uses the so-called 'scientific notation'. Each number is represented as follows:

$$N = (-1)^s \cdot M \cdot 2^E$$

Where, s, M, E are respectively, the sign bit, the 'mantissa' (also called the 'significand') and the exponent. Note that $1 \leq M < 2$.

Now, in the bit-representation of such numbers, the standard for 32-bit floating point numbers is 1 sign bit, 8 (e-)bits for $E + 2^{k-1} - 1$, and the remaining 23 (f-)bits for $M - 1$. Note that, when the e-bits are all 0, the f-bits now represent M as it is. This is called the denormalized representation, while normalized is when the e-bits aren't all 0 or all 1. This distinction is convenient to compare two floating point numbers and allow for greater precision. This becomes clear when looking at the bounds of normalized and denormalized numbers.

Additionally, when the e-bits are all 1, the convention is to use this for very specific values and not to store actually numerical values. When e-bits are all 1 and f-bits are all 0, it is used to represent $\pm\infty$. When e-bits are all 1 and f-bits are not all 0, it is used to represent NaN.

How to not float Obtaining a number in the decimal form from its floating point representation is rather involved, but necessary as we shall see. We denote by e, f the numbers obtained from interpreting the e-bits and the f-bits as in the floating point representation (Note they are not directly E and M). In the denormalized case, *i.e.* when the e-bits are not all zero or all one,

$$\begin{aligned} E &= e - 2^{k-1} + 1 \\ M &= 1 + f \qquad \text{where } f \in [0, 1) \end{aligned}$$

In the normalized case, *i.e.* when the e-bits are all 0,

$$\begin{aligned} E &= 2 - 2^{k-1} \\ M &= f \end{aligned}$$

Here, the $2^{k-1} - 1$ in the calculation of E is called the 'bias' term.

I could be bounded in a nut shell Calculating the largest and smallest possible k-bit normalised and denormalised numbers is a fun exercise (which will soon be added here).

...Operate like a surgeon? Floating point multiplication is extremely straightforward.

Suppose we have

$$(-1)^s \cdot M \cdot 2^E = \left((-1)^{s_1} \cdot M_1 \cdot 2^{E_1}\right) \times \left((-1)^{s_2} \cdot M_2 \cdot 2^{E_2}\right)$$

Then, it is fairly easy to see that

$$\begin{aligned} s &= s_1 \oplus s_2 \\ E &= E_1 + E_2 + c \\ M &= \frac{M_1 \cdot M_2}{2^c} \end{aligned}$$

Where c is the number of right shifts required to ensure $M_1 \cdot M_2$ fits in the 23-bit width in the floating point representation (Okay, a bit too fancy, it's either 0 or 1 :P). If E exceeds the 8-bit limit, then there is an overflow returned.

Now, floating point addition is a bit more involved. Suppose we have

$$(-1)^s \cdot M \cdot 2^E = \left((-1)^{s_1} \cdot M_1 \cdot 2^{E_1}\right) + \left((-1)^{s_2} \cdot M_2 \cdot 2^{E_2}\right)$$

Then, we note that (assuming $E_1 \geq E_2$)

$$\begin{aligned} E &= E_2 + c \\ (-1)^s \cdot M &= \frac{((-1)^{s_1} \cdot M_1) \cdot 2^{E_1 - E_2} + ((-1)^{s_2} \cdot M_2)}{2^c} \end{aligned}$$

Here, s, M are computed using standard signed integer align and addition. Then, c is again the number of shifts required to fit M in the bit limit (now this isn't limited to 0 and 1). Again, if E exceeds the bit limit, then an overflow is returned.

4 SimpleRISC Assembly

Structure SimpleRISC uses 16 registers labelled $r_1, \ldots, r_{15}$, with special names for r_{14}, r_{15} as sp, r_a (stack pointer and return address) respectively. Additionally, SimpleRISC uses two flag registers called `flags.E` and `flags.GT`, which indicate the result of the `cmp` instruction.

4.1 Notation and syntax

Transfer these registers please As a form of notation, we often use register transfer notation. It's a neat way of showing the change in values stored in memory. For example:

$$\begin{aligned} r_1 &\leftarrow r_2 \\ r_3 &\leftarrow 5 \\ r_4 &\leftarrow r_1 + r_2 \end{aligned}$$

Please note, for ease of typesetting, we use r_i to denote the register ri in register transfer notation.

In addition to using register names like r_1 and immediates like 5, we can also refer to the memory location whose address matches the contents of a register like $[r_1]$ This allows us to deal with memory offsets as well, for example: $20[r_1]$ basically referring to $[r_1 + 20]$, that is the memory location 20 bytes from the location stored in the register r_1.

Programmers assemble! The command structure of Assembly programming in general is rather simple:

```
.label:
    ins   arg1, arg2, ...
    ins   arg1, arg2, ...

.label:
    ins   arg1, arg2, ...
```

The instructions possible are as follows (the second argument onwards can be replaced by immediates or memory addresses instead of registers too):

- Moving instructions
  - `mov r1, r2` stores the value of r_2 in r_1.
- Arithmetic and logic
  - `add r1, r2, r3` for $r_1 \leftarrow r_2 + r_3$
  - `sub r1, r2, r3` for $r_1 \leftarrow r_2 - r_3$
  - `mul r1, r2, r3` for $r_1 \leftarrow r_2 \cdot r_3$
  - `div r1, r2, r3` for $r_1 \leftarrow \frac{r_2}{r_3}$
  - `mod r1, r2, r3` for $r_1 \leftarrow r_2 \,\%\, r_3$
  - `cmp r1, r2` compares the values in r_1, r_2 and updates the flag registers accordingly.
- Bitwise instructions
  - `and r1, r2, r3` for $r_1 \leftarrow r_2 \,\&\, r_3$
  - `or  r1, r2, r3` for $r_1 \leftarrow r_2 \,|\, r_3$
  - `not r1, r2` for $r_1 \leftarrow \neg r_2$
- Shift instructions
  - `lsl r1, r2, r3` for storing in r_1 the value of r_2 shifted left r_3 times with 0s appended.
  - `lsr r1, r2, r3` for storing in r_1 the value of r_2 shifted right r_3 times with 0s appended.
  - `asr r1, r2, r3` for storing in r_1 the value of r_2 shifted right r_3 times with the initial bit repeated (essentially dividing the signed representation by 2).
- Load and Store instructions (these are used for base-offset address notation)
  - `ld  r1, 10[r2]` for $r_1 \leftarrow [r_2 + 10]$
  - `st  r1, 10[r2]` for $[r_2 + 10] \leftarrow r_1$
- Branch instructions
  - `b .label` to branch unconditionally to `.label`
  - `beq .label` to branch to `.label` if `flags.E` $= 1$.
  - `bgt .label` to branch to `.label` if `flags.GT` $= 1$.
- Function handling
  - `call  .label` to
- Nope
  - `nop` instruction that does nothing.

Note that immediates are interpreted as 16-bit integers while they are stored as 32-bit integers. Thus, the `u,h` modifiers after the commands can determine whether we are dealing with the least or most significant 16 bits respectively.

4.2 Operation encoding

Each instruction is encoded in a 32-bit string. Usually, it is the first 5 bits reserved for the operation code or *opcode* of the instruction and the remaining 27 bits are for the arguments. The argument encoding varies based on the number of arguments of the instruction itself.

0-address instruction These are either `nop` or `ret`. These understandably don't need anything in the 27 argument bits, so it doesn't really matter :P.

1-address instruction These are `call, b, beq, bgt`, all of which require one argument that is the label to branch to (or call). The 27 bits store what is called an 'offset', which refers to the number of units away from the program counter we need to branch to. Since the offset points to a 4-byte word address, the actual address can be obtained using $pc + 4 \cdot \text{offset}$.

2-address instruction These are `cmp, not, mov`. All of these simply use the 3-address format without using one of the addresses. Simple enough :P.

3-address instruction This corresponds to all the other instructions. They are of the form `ins r1, r2, r3`, where r_3 is either a register or an immediate. The 27 argument bits consist of:

- One I bit which determines whether r_3 is an immediate or a register, storing 1 or 0 respectively.
- 4 bits each to store r_d, r_{s_1}, r_{s_2}, which are the destination and two source registers respectively. There are 16 registers so we only need 4 bits to identify one.
- If $I = 1$, then instead of 4 r_{s_2} bits, there are 2 modifier bits to determine whether there is a `u/h` modifier and then 16 immediate bits.

Back 2-address `cmp` instructions drop the r_d bits because semantically it doesn't make sense to define a destination register for the comparison instruction. Similarly, the `not, mov` instructions use the r_d bits but drop the r_{s_1} bits because there needs to be a destination and source register.

Interesting instructions Now, the load and store instructions behave somewhat interestingly. `ld rd, imm[rs1]` behaves similarly to a 3-address instruction in the immediate form. `st rs2, imm[rs1]` can't really define a destination register semantically, and needs two source registers and an immediate. So an exception is made and r_{s_2} is treated as though it is a destination register and finally uses a 3-address encoding.

4.3 Recursion in programming

When writing a recursive program, or even one with multiple nested function calls, one has to make use of the stack determined by *sp* in order to store values needed to be retained as well as the return address as in r_a since the `call` instruction overwrites r_a. This is not very intuitive initially, but the factorial program as mentioned in the Quiz 1 paper should provide an intuition.

5 ARM Assembly

Structure This is different from SimpleRISC in a number of different ways. Getting the trivial stuff out of the way, immediates are identified with the # symbol before the number. Naturally ARM also uses 16 different registers labelled $r_1, \ldots, r_{15}$ with the last 5 having specific functions. They are in order: fp, ip, sp, lr, pc: the frame pointer, IPC scratch register, stack pointer, link register and program counter respectively.

There is a CPSR (Current Program State Register) which stores a set of flags. The flag set used is far more comprehensive than that of SimpleRISC and that leads to some interesting conditions. The flags and their corresponding semantics are as follows:

Flag	Semantics
Z	Zero (1 if the operation or comparison results in a zero)
N	Negative (1 if the operation or comparison results in something negative)
C	Carry (1 if there is a carry involved in the operation or comparison)
V	Overflow (1 if there is an overflow in the operation or comparison)

Note that the carry flag can be set if there is an overflow in unsigned addition. For logical shifts, the C flag is set to whatever bit was booted due to the shift.

5.1 Notation and Syntax

Instructions are divided into categories which, as we shall see, make encoding a lot simpler. The divisions given below aren't exactly the categories, but are similar.

- Moving, arithmetic and logic
  - `mov r1, r2` does $r_1 \leftarrow r_2$
  - `mvn r1, r2` does $r_1 \leftarrow \neg r_2$
  - `add r1, r2, r3` does $r_1 \leftarrow r_2 + r_3$
  - `sub r1, r2, r3` does $r_1 \leftarrow r_2 - r_3$
  - `rsb r1, r2, r3` does $r_1 \leftarrow r_3 - r_2$
  - `and r1, r2, r3` does $r_1 \leftarrow r_2 \,\&\, r_3$
  - `eor r1, r2, r3` does $r_1 \leftarrow r_2 \oplus r_3$
  - `orr r1, r2, r3` does $r_1 \leftarrow r_2 \mid r_3$
  - `bic r1, r2, r3` does $r_1 \leftarrow r_2 \,\&\, \neg r_3$
  - `mul r1, r2, r3` does $r_1 \leftarrow r_2 \cdot r_3$
  - `mla r1, r2, r3, r4` does $r_1 \leftarrow r_2 \cdot r_3 + r_4$
  - `smull r1, r2, r3, r4` does $r_1 r_2 \leftarrow r_3 \cdot_s r_4$ (Signed multiplication stored in 64 bits)
  - `umull r1, r2, r3, r4` does $r_1 r_2 \leftarrow r_3 \cdot_u r_4$ (Unsigned multiplication stored in 64 bits)
  - `sdiv  r1, r2, r3` does $r_1 \leftarrow r_2 \div_s r_3$
  - `udiv  r1, r2, r3` does $r_1 \leftarrow r_2 \div_u r_3$
- Comparison
  - `cmp r1, r2` sets the flags based on the computation of $r_1 - r_2$.
  - `cmn r1, r2` sets the flags based on the computation of $r_1 + r_2$.
  - `tst r1, r2` sets the flags based on the computation of $r_1 \,\&\, r_2$.
  - `teq r1, r2` sets the flags based on the computation of $r_1 \oplus r_2$.

  Additionally, some arithmetic instructions can have the `s` suffix and be used to set the CPSR flags based on their output as well.
- Load and store
  - `ldr r1, [r2]` does $r_1 \leftarrow [r_2]$.
  - `str r1, [r2]` does $[r_2] \leftarrow r_1$.

  Bonus: we can add suffixes `b, h, sb, sh` to load and store single bytes, two bytes, a single signed byte and two signed bytes respectively.
- Branching
  - `b   .label` branches unconditionally to `.label`.
  - `bl  .label` branches unconditionally to `.label` and does $lr \leftarrow pc + 4$.
  - `bx  r1` branches unconditionally to the address contained in r_2.

  The `b` instruction can be suffixed by any of the 15 conditional codes to make it a conditional branch.

Incidentally, the last argument in a lot of the above instructions can be either a register, an immediate (prefixed by #) or a scaled-index-notation based memory address. These scaled index memory addresses are achieved using `[r1, r2, #2]` which would be referencing $[r_1 + (r_2 \ll 2)]$

5.2 More advanced shenanigans

We also have instructions to specifically update the stack when spilling registers for a recusrive function call. The commands `ldmfd sp!, {r1, r2, ...}` and `stmfd sp!, {r1, r2, ...}` are used to load and store the values of registers from the values stored in the stack. The `!` after the stack pointer indicates that the stack pointer gets updated accordingly after the corresponding operations. Thus, it can also be omitted. The register order inside the braces does not matter. They are always stored and accessed in order.

5.3 Instruction encoding

The categories that the instructions are divided into come into play here, as the encoding principles are different for different categories.

Data processing instructions These consist of moving, arithmetic, logic and comparison instructions. These are encoded in 32-bit strings as follows:

- The first 4 bits are for the conditional suffix if any.
- The next two are always 00 to indicate that it is a data processing instruction. (These are called the type bits)
- Next is the I bit which indicates whether the final argument is an immediate or a register. (1 and 0 respectively)
- Next are 4 operation code bits corresponding to the 16 base data processing instructions.
- The next bit is the S bit which if on will update the CPSR based on the operation outcome.
- The next two sets of 4 bits are for the source register and destination register respectively.
- The final 12 bits are for the second source register or immediate value. (The way a 16 bit immediate is encoded in 12 bits will be elaborated on soon)

Load/Store instructions The general encoding schema is:

- 4 conditional bits followed by two type bits which are always 01.
- 6 bits representing I, P, U, B, W, L, I being the same as usual, P being the sort of indexing to use (pre- or post- as 1 and 0), U being whether to subtract or add the offset to the base (0 and 1), W being whether to even use pre- or post-indexed addressing and L being whether it is a load or store instruction (1 or 0)
- 4 bits each for the source and destination registers
- 12 bits for the shifter operand or immediate

Branching instructions The encoding schema for such instructions is:

- 4 conditional bits (very important here)
- Two type bits (which are 10 in this case)
- One always set bit, followed by the L bit which indicates whether the branching should include linking.
- 24 offset bits to calculate how far from pc to move.

Gimme 12 bits immediately! Since there are only 12 bits left for immediates in almost all the instructions, we must find an efficient way to encode as many feasible values as possible for the immediates in 12 bits. To do so, we reserve 4 bits called the 'rotation' bits (r, say) followed by a byte for the value to store. The value is then right-rotated by $2 \cdot r$ which gives us the value in the immediate.

Shifty lil bugger Shifter operands are also rather messy to encode. Now for this section we shall number the bits from right to left instead of left to right.

6 x86 Assembly