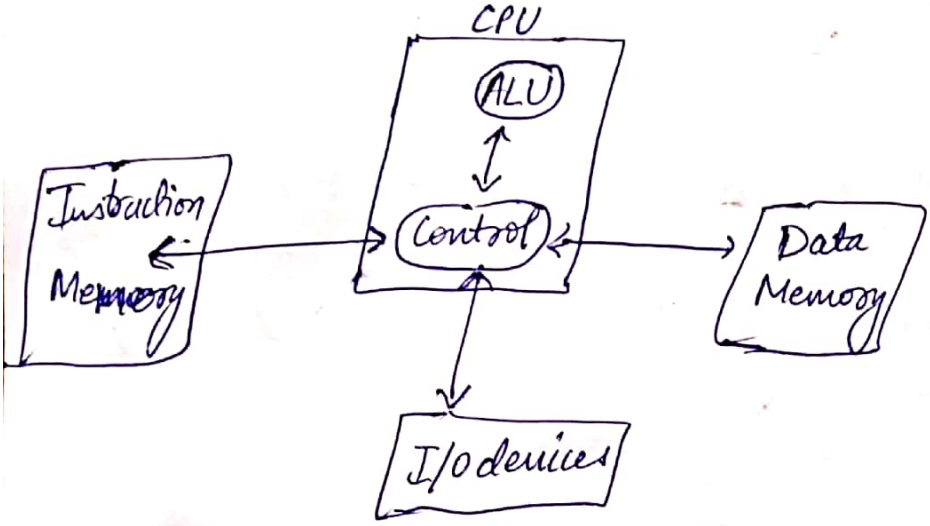
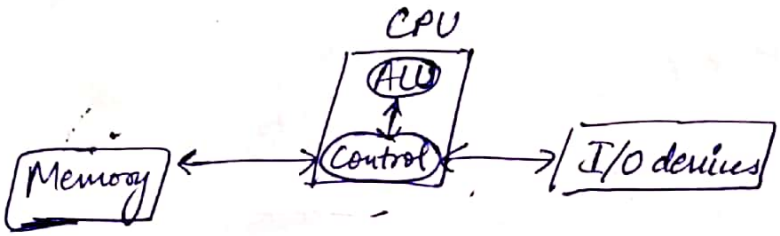
Computer Architecture & Organisation

Chapter – 1 → Harvard Architecture



Von Neumann Architecture



← It uses the same memory for instructions & data by encoding them into symbols normally used in memory. The CPU decodes every instruction. This is the stored-program concept.

Modern machines also use registers (in CPU storage locations), stacks (no operands reqd. for instructions), accumulators (intel instructions sometimes use registers like eax as accumulators)

Chapter – 2 → Floating point numbers

Sign(S)	Exp (E)	Mantissa(M)
1	8	23

The number $A = (-1)^S \times P \times 2^{E-bias}$ $\quad 1 \le E \le 254$

$E = X + bias$ where $bias = 127$ & X is actual exponent of number. So $X = 0$ is represented by $E = 127$

$P = 1 + M, \quad 0 \le M < 1$

Mantissa is written from after point. Number is assumed to be of the form $1.\ldots \times 2^{+X}$. E varies from ~~0 to 255~~ 1 to 254. E = 0 & E = 255 are reserved.

E	M	Value
255	0	∞ if S=0
255	0	−∞ if S=1
255	≠0	NAN
0	0	0
0	≠0	Denormal Number

← There are 2 representations of 0, for S=1 & −1.

Denormal Numbers → introduced to give more range for small numbers.

$$A = (-1)^S \times P \times 2^{-126} \quad (P = 0+M,\ 0 \le M \le 1)$$

Exponent is −126 by default, so smallest denormal number is

$$0.\underbrace{00\ldots01}_{22} \times 2^{-126} = 2^{-149}$$

largest is

$$0.\underbrace{111\ldots11}_{22} \times 2^{-126} = 2^{-126} - 2^{-149}$$

largest denormal number is smaller than smallest positive normal number.

* Floating point operations are commutative but not associative. Order can play a role when dealing with high precision.

Chapter–3 → Assembly language refers to a family of low-level programming languages that are specific to an ISA. Each statement has an instruction code that translates to a basic machine instruction & a list of operands.

Basics of Assembly → **Machine Model** → Von Neumann machine augmented with registers

CPU reads out program's instruction-wise & executes them. The program counter (PC) keeps track of the instruction being executed. Instructions get operands from main memory & registers. CPU co-ordinates everything.

Registers → ~~Languages~~ languages give registers for use.

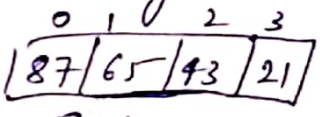
r0 ... r15 in ARM, eax, ebx, ecx, edx, esi, edi, ebp & esp in x86 (32 bit). Most ISA use a return address register except x86. ARM gives access to PC in r15. x86 keeps it implicit & hidden.

Memory → Address is a unsigned 32-bit integer in 32-bit machines & 64 in 64-bits. Memory can ~~et~~ only be accessed as a byte. For multi-byte data like short (2 bytes), int (4), long (8), there are 2 representations

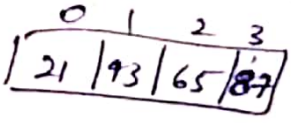
Big Endian → Store MSByte in lowest address.

Little Endian → Store MSByte in highest address.

0x87659321 →



Big endian



Little Endian

x86 is little-endian, ARM was as well, now supports both. IBM Power & Sun SPARC processors are big-endian.

Representing arrays → Stored in contiguous locations. A multi-dimensional array is stored by flattening out other dimensions. Row-major representation is when an array is stored row-wise in memory i.e. each row is stored as a contiguous block. Same for column-major.

Assembly - language Syntax → for GNU assembler.

File structure → There are many sections, they start with heading . sectionname . Common sections are .text(actual code), .data (Initialised data), .bss (data initialised to 0), . file (at start of file to specify name of file)

Basic statements → Instruction operand1 operand2 ....
The list of operands has either the value or location of the operand. The value is a numeric constant & is called an immediate value.
An assembly statement can have a label, instruction & a comment.

Types of instructions → On basis of functionality:-
(1) Data processing (2) Data transfer (3) Branching (4) Exception generating

On basis of no-of operands:-
↳ If an instruction requires n operands, it is a n-address format instruction
In ARM, data processing instructions → 3 address format
data transfer instructions → 2 address format
X86, most are 2-address format.

Types of operands → Method of specifying & accessing an operand in assembly is known as addressing mode.

immediate-addressing → specify integer constants as operand in the instruction

Register-transfer notation → $r_1 \leftarrow r_2 + r_3$
Add contents of r_2 & r_3, store it in r_1

$r_1 \leftarrow [r_2]$
Fetch content of the memory address stored in r_2 & store it in r_1.

Generic addressing modes for operands :— Represent value of operand by V

immediate → V ← imm (Use constant imm as value of operand)

register → V ← r_1 (Processor uses value in register r_1)

register_indirect → V ← [r_1]

base-offset → V ← [r_1 + offset] (offset is a constant. Also called displacement)

base-index → V ← [r_1 + r_2] (r_1 is base register, r_2 is index register)

base-index-offset → V ← [r_1 + r_2 + offset]

memory direct → V ← addr (Value is contained at address addr which is a numerical constant. Memory address is directly embedded in the instruction)

memory-indirect → V ← [[r_1]]

PC-relative → V ← [PC + offset] (useful for branch instructions)

↳ Memory address specified by an operand is called effective memory address.

SimpleRisc ISA → 21 instructions.

Machine model → 16 registers r_0, r_1 --- r_{15}. 14 general purpose
r_{14} → stack pointer (SP), r_{15} → return address (ra)
32-bit registers. Internal register called flags. flags.E (equal), flags.GT (greater than). E is set if comparison gives equal. GT is set if first operand > second operand.

↳ Each instruction is encoded into 4 bytes/32-bits in memory.

Register Transfer Instruction :- mov reg, (reg/imm)

Arithmetic Instructions :- add, sub, mul, div, mod

for all of them → op reg reg (reg/imm) (ex. add r1 r2 r3)

cmp reg, (reg/imm) (sets flags by comparing)

Logical Instructions :- ~~and~~ compute bitwise AND, OR, NOT.

and reg reg (reg/imm)
or reg reg (reg/imm)
not reg (reg/imm)

(not r1 r2 ⇒ r1 ← ~r2)

Shift Instructions → ~~left~~ logical shift left (lsl), logical ~~right shift~~ shift right (lsr) arithmetic shift right (asr).

	Ex
lsl reg, reg, (reg/imm)	lsl r3 r1 r2 ⇒ r3 ← r1 ≪ r2 (shift left)
lsr reg, reg, (reg/imm)	lsr r3 r1 r2 ⇒ r3 ← r1 ≫ r2 (shift right & fill with zeros)
asr reg, reg, (reg/imm)	asr r3 r1 r2 ⇒ r3 ← r1 ≫ r2 (shift right & fill with ~~0~~ MSB)

Data-transfer Instructions :- load (ld) to load instructions from memory into registers, store (st) for the opposite.

	Ex	
ld reg, imm[reg]	ld r1, 10[r2]	r1 ← [r2+10]
st reg, imm[reg]	st r1, 10[r2]	[r2+10] ← r1

Unconditional Branch instruction:

	Example
b label	b .foo ⇒ branch to .foo

Conditional Branch instructions :- branch based on the flag register.

	example	
beq label	beq .foo	branch to .foo if flags.E = 1
bgt label	bgt .foo	// // // // flags.GT = 1

Calling & Returning from Functions → If only a small number of values have to be passed b/w caller & callee, then use registers. Otherwise use memory. To prevent the callee overwriting registers the caller may require, either the caller saves them in memory before calling or the callee saves & restores them before returning. Saving registers in memory is called register spilling.

To solve all problems of function calls, we use a stack. A stack starts at some high location in memory & grows downward. It ~~contains~~ maintains the activation blocks of the functions in a stack. To return a large thing, callee writes it to activation ~~block~~ block # of caller.

~~Activation block~~ Activation block of a function

arguments
return address
register spill area
local variables

(Top of stack is at lower address than bottom)

The callee creates a new activation block ~~before calling the function~~ by doing these steps:-

1. Decrement stack pointer by size of activation block.
2. Copy values of arguments.
3. Initialize local variables to memory if reqd.
4. spill registers (including ra) if reqd.

The activation block is destroyed when function returns by adding its size to sp

This solves all our problems. Caller & callee dont overwrite each other's ~~activation~~ local variables, it is possible to stop register overwriting by specifying instructions to save them in activation blocks. No need to have an explicit agreement about memory to pass arguments, if caller just pushes them on stack & callee uses them. Similarly to return values, callee first destroys its stack by resetting sp. Then pushes values on stack which caller can use. If the callee calls another function, it can spill ~~its~~ its ra in its activation block & restore it later.

call & ret → call label | example: call .foo | explanation: ra ← PC+4; PC ← address of .foo

ret | ret | PC ← ra

ret is a 0-address instruction. call & ret can be thought of as unconditional branch instructions coz they change PC.

---

nop → does absolutely nothing. (All 21 ~~the~~ instructions done)

Modifiers → You can't load a 32-bit constant when the whole instruction is 32-bit. So you have to load the upper & then lower half seperately.

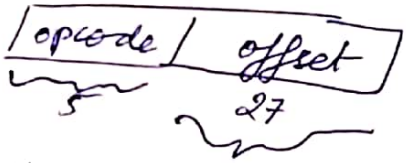
Add suffix 'u' or 'h' to instructions except shift instructions. By default, the processor performs sign extension when converting 16-bit constants to 32-bit. 'u' specifies it to treat it as an unsigned constant & append only zeros. 'h' tells it to load it to the upper half of the register directly. So, a constant 0xFB12CDEF can be loaded into r0 in 2 instructions.

```
movh  r0, 0xFB12
addu  r0, r0, 0xCDEF
```

Encoding the SimpleRisc ISA :- We have to encode each instruction to 32-bits. Instructions are in 0, 1, 2, 3 address formats. There are 21 instructions, so we need 5-bits to represent them. Code for an instruction is called its opcode.

0-address → ret & nop → only need 5-bits for opcode.

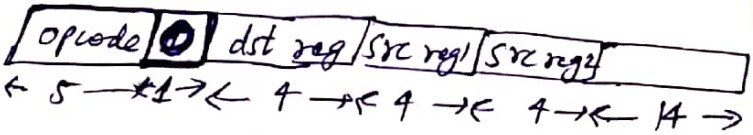
1-address → call, b, beq, bgt → take label as argument which is an address in memory. We only have 27-bits but address is 32-bits. Here we use PC-relative offset representation. So, we can represent addresses $PC \pm 2^{26}$. Assume instructions are 4-byte aligned. Then every offset has 00 as last 2 bits. So we make them implicit. So, we can represent addresses $PC \pm 2^{28}$.



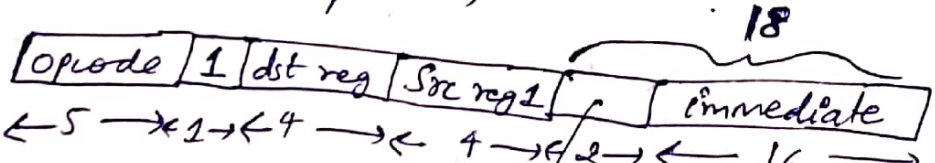
Encoding of 1-address instruction.

3-address → need 1 bit to specify if third operand is register or immediate value, 16 registers so need 4-bits for each register

I=0 ⇒ register operands, then



I = 1 ⇒ immediate operand, then

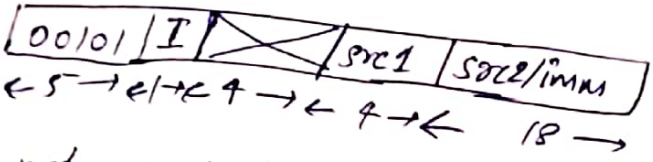


2 modifier bits
00 ⇒ default
01 ⇒ u
10 ⇒ h

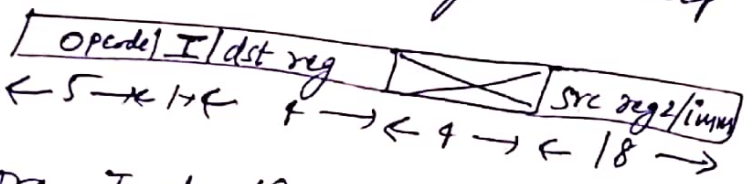
Processor internally expands the immediate to a 32-bit value according to the modifier bits.

2-address instructions → We will use the 3-address here as such so that the processor decode logic is simpler

cmp → has 2 source-operands. keep dst-rg empty.

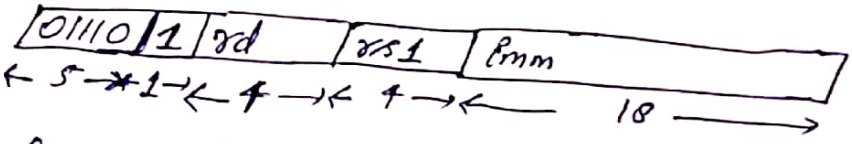


mov & not → 1 dst reg & 1 src operand that can be either immediate or register. keep src1 empty.



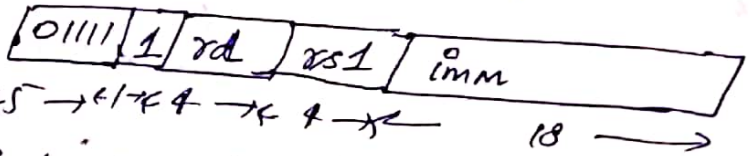
Load & Store Instructions: ld & st are 2-address instructions. The second operand is a base register & a constant offset using base offset addressing. For load, we use the 3-address format in immediate mode

ld rd, imm[rs1] →



Store instruction. doesnt have a destination register but a memory location. We still use the same format but the destination register argument ~~would be the dest~~ would be the source actually. We dont want a new format for just 1 instruction. We choose efficiency over elegance.

st rd, imm[rs1] →



* So, there are total 3 instruction formats, branch, immediate & register (ret & nop are special case of branch)