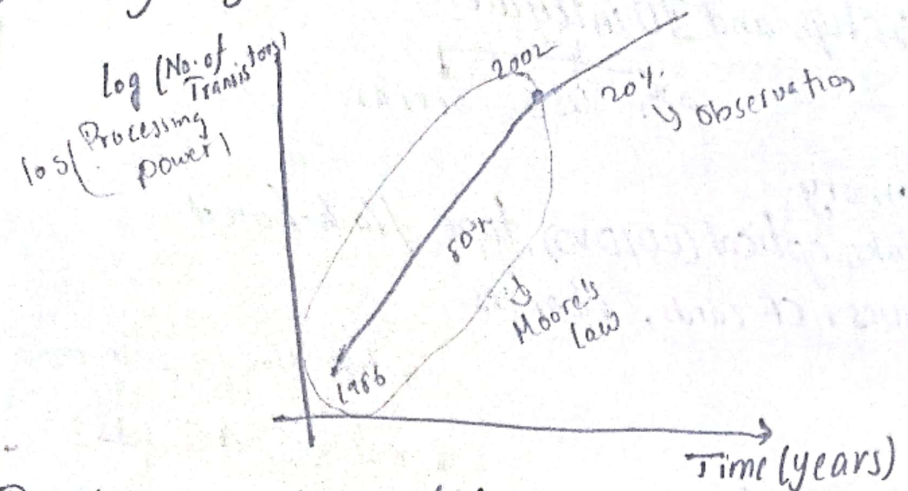
Computer Architecture

Moore's law: Observation (Pseudo law) that number of transistors in an IC doubles every 2 years.



Processing power increased by 50% every year until 2002, then 20% because of over heating (max power consumption limit).

Hierarchy of Languages

Algorithm
↓ Programmer
High Level Language } Machine Independent
↓ Compiler
Assembly Language
↓ Assembler
Machine Language } Machine Specific
↓ Computer designer
Digital Logic

Von Neumann Architecture (Stored program concept)

5 components
- Input
- Output
- Memory
- Datapath } processor
- Controlpath

Computer
- IC /chips → CPU (processor), main memory, cache memory
- Motherboard → chips and IO interfaces (PCI, USB, Serial)
- Secondary Memory
  - Magnetic disks, optical (CD/DVD), tape, flash-based USB pendrives, CF cards, floppies

IC Technology

IC cost

$$\text{Cost per die} = \frac{\text{Cost per wafer}}{\text{Dies per wafer} \times \text{yield}}$$

$$\text{Dies per wafer} \approx \frac{\text{Area of wafer}}{\text{Area of die}}$$

$$\text{Yield} = \frac{1}{[1 + (\text{Defects per area} \times \text{Area of die})]^2}$$

no. of transistors/chip increased

SSI, MSI, LSI, VLSI, ULSI
small-scale Integration ... ultra large

unit cost of chip decreases with volume of production

→ Physical dimension limit

→ Power consumption limit
- fan sinks heat
- freq of clk reduced based on usage

Instruction Set design

Instruction set → interface between hardware and software.
↓
defines a machine

Processor — 1) Fetches instruction from memory
2) Executes instruction from memory } loop

Eg: a := b + c
- a, b, c → operands
- addition (+) → operation

MIPS → 32 registers
- each of 32 bits (word)
- 4 bytes
- 2 bytes → half word

MIPS Instructions

1) add <res>, <op1>, <op2>
   Eg: add $S3, $S2, $S1

2) addi <res>, <op1>, <const>
   Eg: addi $S2, $S1, 100

3) lw <dst-reg>, <offset>(<base_reg>)
   Eg: $S1, 4($S0)
   ↳ $S0 stores an address

4) sw <offset>(<base-reg>) <src-reg>
   Eg: 12($S0), $S5
   ↳ must be multiple of 4

5) sub
6) mul R0 Rs Rt
7) muli
8) sll (<<)
9) srl (>>)
10) and, andi (&)
11) or, ori (|)
12) nor (~(not))
13) beq <reg1>, <reg2>, <branch-tgt>
    (branch if equal to)
14) bne (branch if not equal)
15) j <jump target>
    ↳ label in assembly
    ↳ offset in machine
16) slt <dst>, <reg1>, <reg2>
    (set less than) (if reg1 < reg2 → dst = 1, else 0)
17) slti
19) addu, addiu (unsigned addition), subu, sltu, sltiu
19) mult Rs Rt ↳ do need to use HI, LO, multu
    ↓ div, divu
20) MFHI, MFLO (move from high / low)
    MTHI, MTLO (move to)
    Eg: MFHI $S2 ↳ $S2 = $HI
    div → R Q
    mul → higher 32 bits, lower 32
21) jal, jalr, jr ↳ Eg: jr $ra

V- return
a- args
sp- stack ptr
gp- global ptr
k- kernel
ra- return address

MIPS registers:
$at, $v0, $v1, $a0-$a3 (1, 2, 3, 4-7)
$s0-$s7 are 16-23 registers
$zero → 0th register (always 0)
$t0-$t7 (temp regs) (8-15)
$t8, $t9 (24, 25), $k0, $k1 (26, 27)
gp, sp, fp, ra (28, 29, 30, 31)

Program Counter (PC)

- 32-bit reg pointing to curr instr
- PC += 4 (if no special instruction to jump)
  ↳ and in 32 regs

Pseudo instructions

move, li, la, blt, bge, bgt, ble

lw can be a Pseudo inst if used as
lw $S1 LABEL

BGTZ, BGEZ, BLEZ, BLTZ, BNEZ (branch if ... zero)

sgt, sge, sle

Load and Stores

LB, LBU, LH, LHU, LW

LBU, LHU: unsigned
LB: the MSB is extended for the higher bits
LH: MSB is extended for the higher bits
LW: 0 is extended for the higher bits

SW, SB, SH

Week-2

Instruction Encoding

32 bit encoding

3 types

1) R-type (reg-reg operations)

opcode	rs	rt	rd	shamt	funct
6	5	5	5	5	6

2) I-type :- loads, stores, addi, muli, etc, bne, beq, jr, jlr

opcode	rs	rt	imm / const / offset rel to PC+4
6	5	5	16

3) j-type

opcode	offset rel to PC+4
6	26

j, jl, trap and return

Function Call Support

jal → jump and link

jal F_LBL

↳ 1) $ra = PC+4
   ↳ One of MIPS reg
2) control to F_LBL

In the function F

:

jr $ra

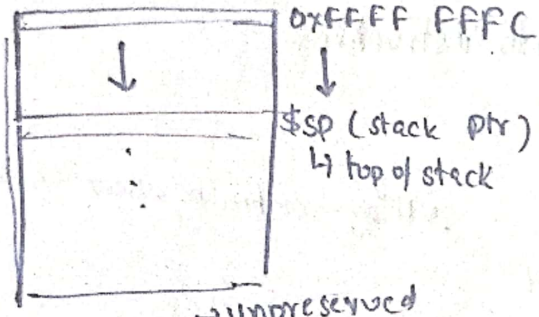
↓ jump reg

$a0, $a1, $a2, $a3 → args

$v0, $v1 → return

Nested calls → Use stack type memory

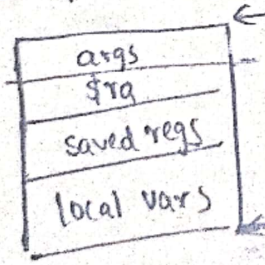
Memory (Stack goes down)



Caller saved regs (unpreserved) : $a, $v, $t → to be saved by callee if cares

Callee saved (preserved) : $ra, $s, $sp, $fp → must be saved restored by callee

Frame pointer ($fp - MIPS32) → easier use



← $fp ↳ easier to use (offsets)

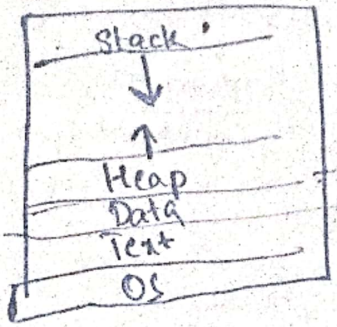
will not fp → ref will be same

but sp → changes with change in sp

and easier to pop the stack

← $sp

Memory Organisation



$gp → points to global vars in data section

← $PC (program memory)

Pseudo Instructions

li — addi/ori

beqz — beq , $zero

~~bnz~~ bnez — bne $zero

~~lb~~ Half words/bytes

lb, lh, lbu, lhu

sb, sh, but no sbu, shu

lb → sign extends the MSB, i.e if 10010010 → 1111 1100100 (24)

lbu → appends 0's → 0000 01001001 (24)

$at → assembler temporary → manages pseudo instructions

linker's

HLL code to Running

GNU gcc / g++ → compiler, assembler, linker

Assembly Language (.s) → Object files (.o)

few .o files → linker → executable → Loaded → run

Machine language loaders

Loading → brings program from secondary memory to main memory and ready for execution

Contents of Object file

1. Header
2. Text segment
3. Static data
4. Relocation Table (list of unresolved inst's) (fn in another file)
5. Symbol Table (table of unresolved symbols) (where vars are loaded into memory)
6) Debugging info

Linker → generate exe files
- Organize text and data as it would appear in memory
- Resolve symbols (memory allocated)
- Rewrite instns referred in reloc table

Loader
- Place text and data in mem
- Init stack, other regs, including $ra

$ra → main

OS loads

Dynamic Linking
- Small exe files better memory
- new libs can be used easily
- Jalr → Function pointers supported

main.s

```
.data Y;
main:
jal F   - I1
lw $s0 X - I2
G:
lw $s1, Y - I3
```

Reloc Table: I1, I2, I3 (for) (no X) (no Y)

Symbol Table: main, G, Y, F

aux.s

```
.data X
F:
jal G   - I4
lw $s2 X - I5
sw Y $s3 - I6
```

Reloc Table: I4 (no G), I5, I6 (addresses of Y in main.exe not determined)

Symbol Table: F, G, X, Y

Boot loader loads OS

OS loads loader

calls loader

Arithmetic in MIPS

MIPS → 2's complement
unsigned representations.

add → addu, addi → addiu, sub → subu, slt → sltu, slti → sltiu,
addu, addi → seperate ovf condition

mult, multu → hi, lo
↓ upper 32 ↓ lower 32

div, divu → hi, lo
↓ remainder ↓ Quotient

hi, lo → not part of 32 MIPS

mfhi <destreg> (moves from hi to destreg)
mflo

Dynamic linking
- Instruction (function call) points to a dynamic linker that resolves the instruction
- jalr → the pointer to func is loaded in reg and jumped to addr in reg

Floating point representation

- Sign, significand, exponent
  - Single Precision → 1 + 8 (exponent) + 23 → significand
  - Double Presion → 1 + 11 (exponent) + 52 → significand

Subnormal numbers → NAN
↳ represent out of bounds

Arith
add.s (floating point), add.d, sub.s, sub.d, mul.s/d, div.s/d
↓ single prec ↓ double prec

Comp
eq, ne, lt, le, gt, ge
c.eq.s, c.le.d.
(6) (2) → 12 cmp instructions

Sets a special bit to 1 if true
bc1t, bc1f → next instruction
↳ takes branch if 1
doesn't take if branch

true / false

CP1 → coprocessor 1 → used to do FPA
in prev implementation f.p.n → complex

lwc1, ldc1, swc1, sdc1
double

MIPS → 32 × (32 bit regs) for FP

↓ can be considered

16 × (64 bit regs) for double precision

Memory operations

lwc1, ldc1, swc1, sdc1

Computer Performance

↳ Execution Time

① Computer Performance equation

Prog. exec. time = #instruction × [#cycles/instru] × [clock-cycle time]

- #instruction: only no of exec programs; can be got from → profiling, → simulators, → hw counters
- #cycles/instru = CPI: can be calc from instruction mix
- clock-cycle time: easily calc from clk freq

$CPI_{avg} = \sum CPI \times f_i$

Types: arithmetic, memory, branch

Factors affecting performance

Factor	Affected aspects
Algorithm	#ins, CPI
HLL	#ins, CPI
Compiler	#ins, CPI
ISA	#ins, CPI, cycle time
H/W impl	CPI, cycle time

Amdahl's Law: Performance improvement is limited by the fraction of program you are improving

F	1-F

→

F/n	1-F

factor of Improvement

$$= \frac{1}{1 + F/n - F} < \frac{1}{1-F}$$

Benchmark, Workload ?

and computer performance analysis

↳ a specific program that measures the CPI

SPEC Benchmarks → System Performance Evaluation Corporation

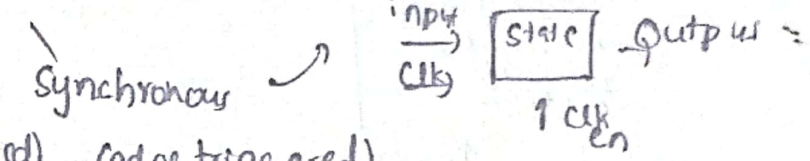
SPEC-2000 → gzip, gcc

↳ collection of programs

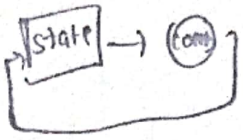
→ Hardware Implementation of MIPS

Sequential circuit

asynchronous (level-triggered) | Synchronous (edge-triggered)



Seq + comb



Setup time (s) :- Time before active edge for which inputs must be held constant

hold time (h) :- The min time inputs to be held const. after active edge

delay time (d) :- time for seq circuit

Then $C > (s + h + d)$

↳ clock cycle time

Data and Control Path

- State elements
- comb. elements
- interconnects

- when to write
- what to write
- state transition

→ Single Cycle Implementation

↳ CPI = 1

Subset of instructions → add, sub, and, or, slt (R-type)

→ lw, sw (I-type)

→ beq (I-type)

→ jump (J-type)

R-type → opcode - 000000 → differentiated by funct

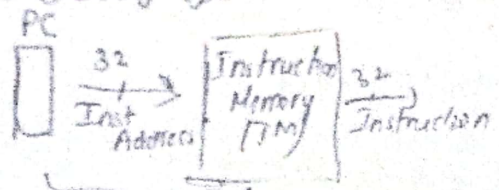
Fetch (inst)

Execute → (1) read register

(2) Instruction specific

(3) PC = PC + 4

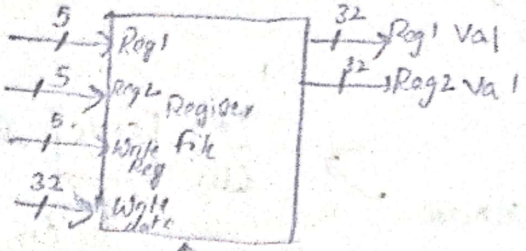
Fetching of Instruction



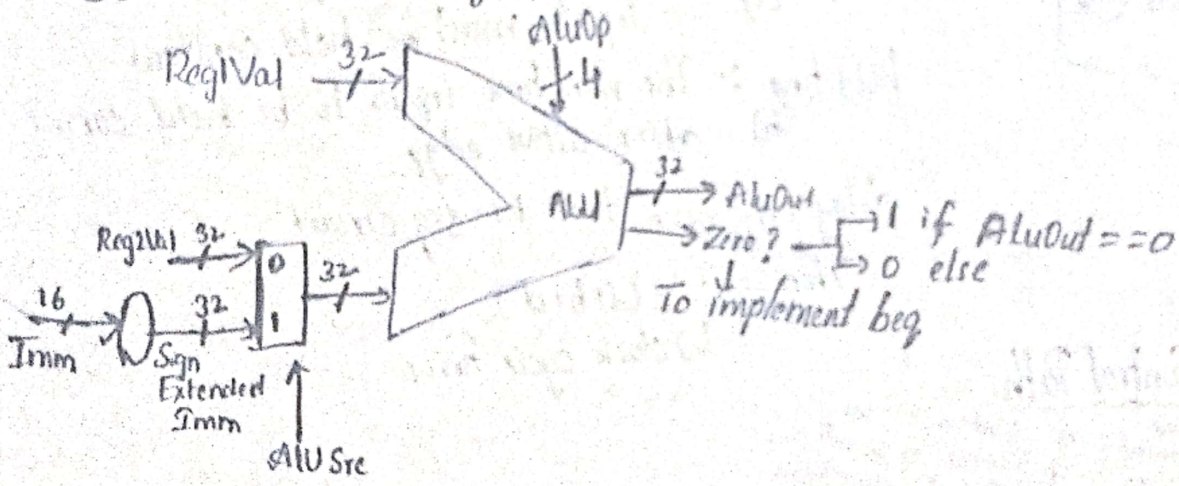
Sequential

lw → only instruction in subset where only one reg is read

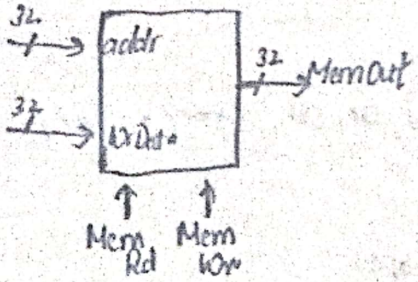
Instruction Execution



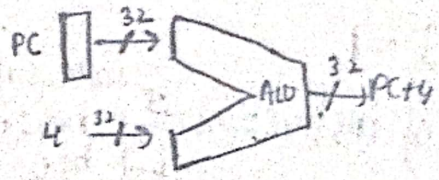
ALU: Arithmetic Logic Unit



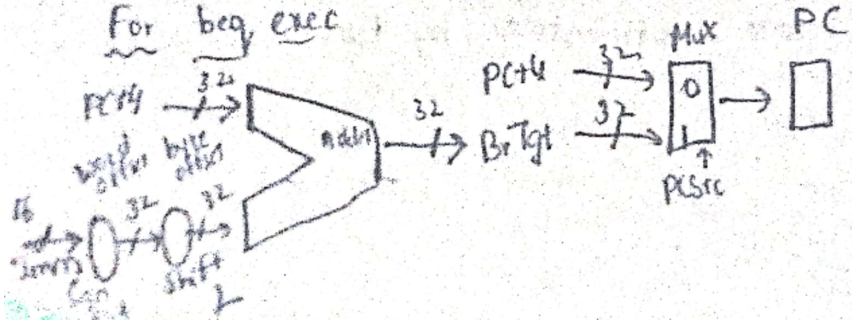
Data Memory

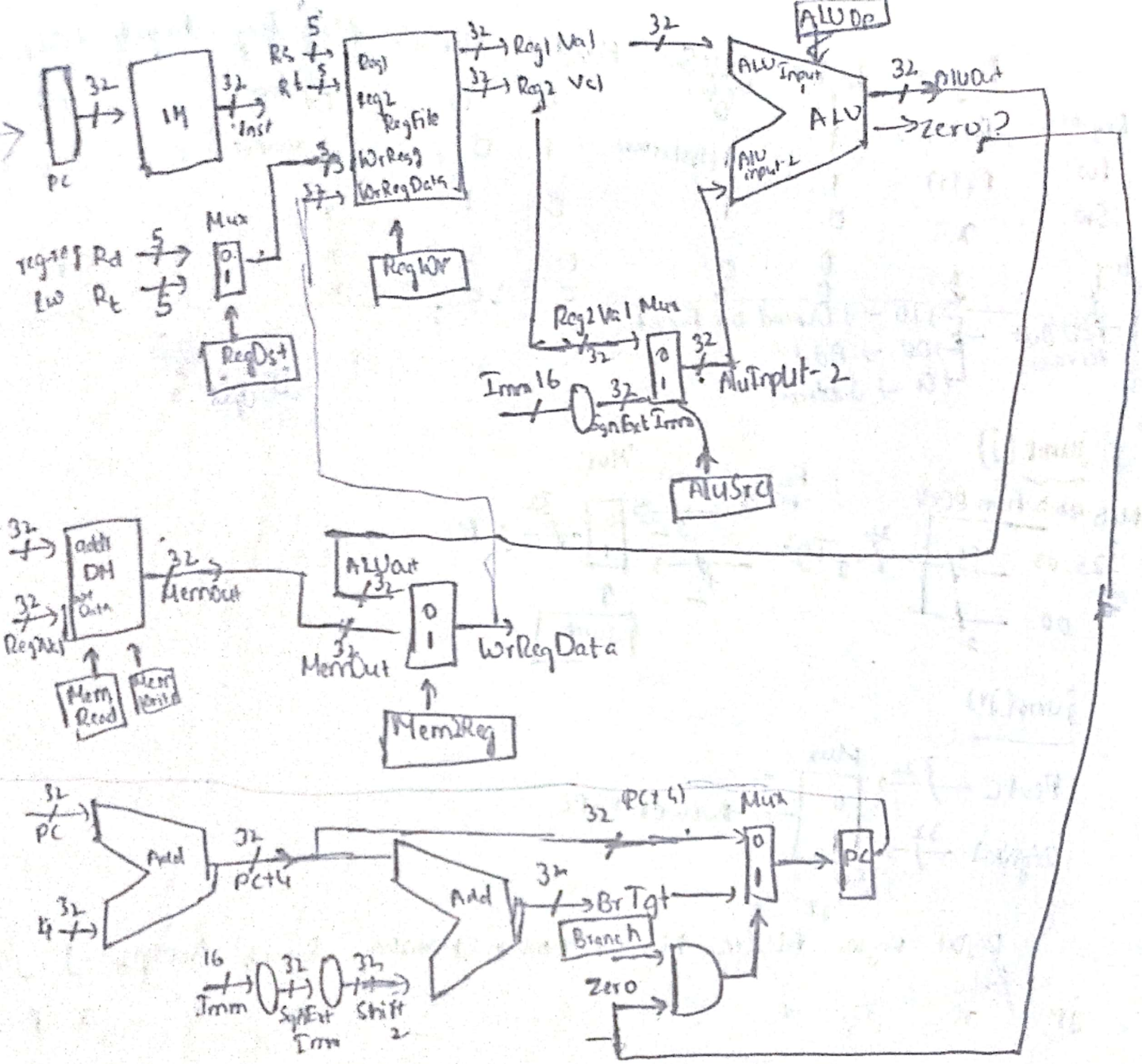


PC computation



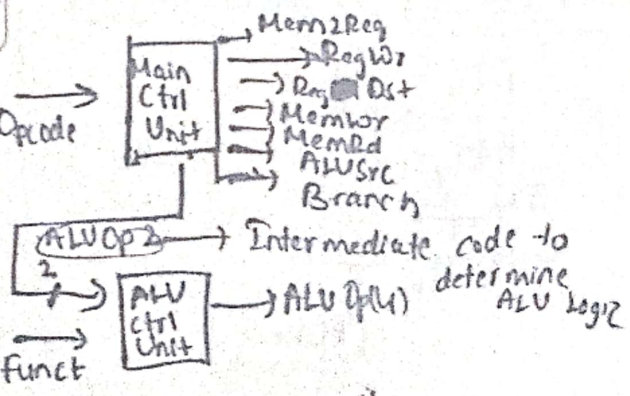
For beq exec





11 bit Control lines in the above diagram

- RegWr (Should Reg be written)
- Reg Dst (Rd vs Rt)
- Mem 2 Reg (ALUOut vs MemOut ?)
- ALUSrc (Imm vs Reg2Val)
- ALU Op(4) (Type of ALU operation)
- MemRd (Mem be Read?)
- MemWr (Mem be Written?)
- Branch (Branch ?)

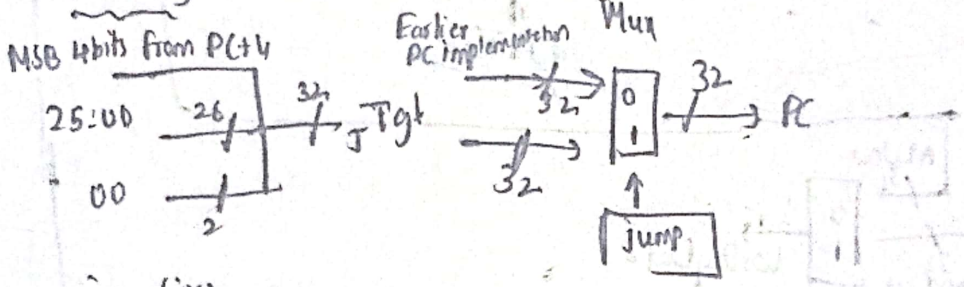


Reason for 2 control units is to generate 1 bit Ctrl lines faster to reduce Clk cycle

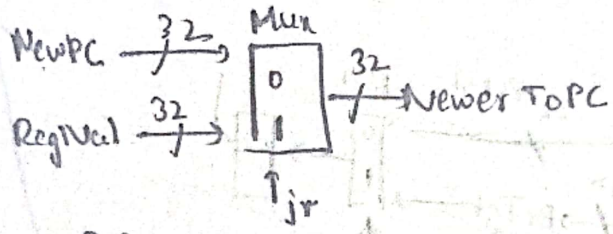
	RegDst	RegWr	ALUSrc (RegVal)	MemRd	MemWr	Mem2Reg	Branch	ALUOp2
Reg-Reg	Rd(0)	1	0	0	0	0 (ALUout)	0	10
lw	Rt(1)	1	1 (SignExtend)	1	0	1 (MemOut)	0	00
sw	x	0	1	0	1	x	0	00
beq	x	0	0	0	0	x	1	01
	x	0	x	0	0	x	x	x

ALUOut (Auxiliary) → 10 → Depend on funct
→ 00 → Add
→ 01 → Subtract

jump (j)

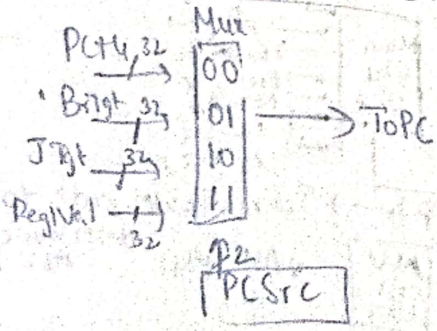


jump(jr)



	RegDst	RegWr	ALUSrc	MemRd	MemWr	Mem2Reg	Branch	ALUOp2	j	jr
jr	x	0	x	0	0	x	x	x	x	1

Alternate DP, CP, Modif to j, jr



The Cycle should be level triggered, since there are dependencies

PC-write is triggered with latter half of cycle to prevent PC writing multiple times in same clk cycle

Reg-write → also on latter half

Mem-wr → also on latter half (efficiency improved only)

→ Clk cycle adjusted for longest instruction

Inefficiencies

→ Cost (lot of hardware)