Lec 14-3-22

ISA

1. Limited memory
2. Slow memory
3. Ease of Programming

↙ 3 address format

Opes^n	operand 1	Operand 2	dest^n
7	32	32	32

↳ 103 bits.

1. → hv to minimize no. of bits per instruction

A = B + C
3000, 3000, 5000 } addresses.

⤷ ADD 3000, 5000, 9000

- cannot throw out "opes^n"

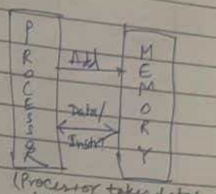
↑ these are "explicit" (explicitly mentioned where the operands, dest^n are...)

for Limited Memory

Implicit operands

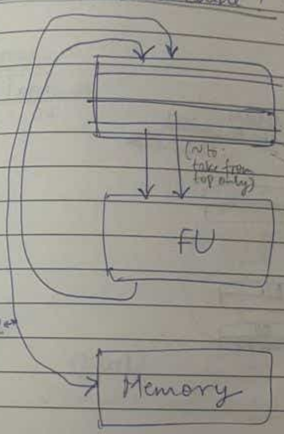
0-address format

operation
7



(Processor takes data/instr^n from memory & does comput^n...)

Can we construct a machine ? Yes!!



LIFO semantics

temp memory (Stack)

take 1 or 2 (unary / binary oper^n) operands from top of stack, & return to top of stack.

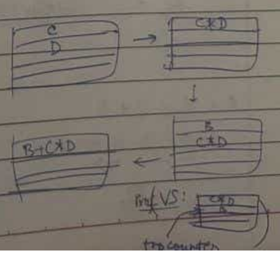
Memory comm^n { PUSH / POP

1 address format
⇓
39 bits

---

e.g. A = B + C * D

PUSH C	39	} Memory ↑ addresses.
PUSH D	39	
MUL	7	
PUSH B	39	
ADD	7	



Lec 14-3-22

ISA

1. Limited memory
2. Slow memory
3. Ease of Programming

3 address format

Opcᵒ	operand 1	Operand 2	destᵒ
7	32	32	32

↳ 103 bits.

↳ 1. → try to minimize no. of bits per instruction

A = B + C
(9000) (3000) (5000) } addresses

↳ ADD 3000, 5000, 9000

- cannot throw out 'opcs'

↑ these are "explicit" (explicitly mentioned where the operands, destᵒ are...)

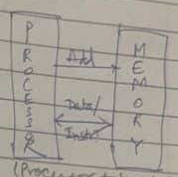
for Limited Memory.

Implicit operands

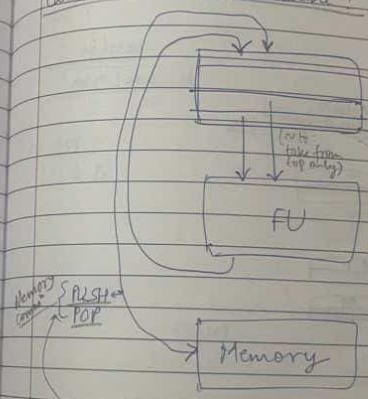
0-address format

operation

Can we construct a machine? Yes!!



(Processor takes data/instr from memory & does computᵒ...)



LIFO semantics

~~temp memory~~ (stack)

take 1 or 2 (many pairing opᵗˢ) operands from top of stack, & return to top of stack.

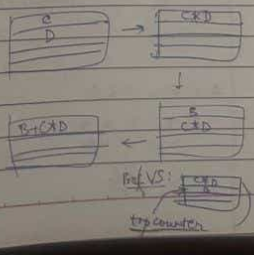
1 address format

⇓

39 bits

A = B + C * D — Memory Addresses.

PUSH	E	39
PUSH	D	39
MUL		7
PUSH	B	39
ADD		7
POP	A	39



Page No.
Date

A = B + C * D + E/F ⇒

just ~~won't work~~

B+C*D

⇓

E/F नहीं...)

PUSH C
PUSH D
MUL
PUSH B
ADD
POP T1
PUSH E
PUSH F
DIV
PUSH T1
ADD

extra (pair) due to limited temp. size

T1

E/F

E/F
T1 = B+C*D

A = B + C
+ BC
BC +

Postfix / Polish notation [* Prof pages — Algo — Hack...]

A = B + C * D
= BCD * +
⇓
NB

Page No. 175
Date

PUSH B
PUSH C
PUSH D
MUL
ADD
POP A

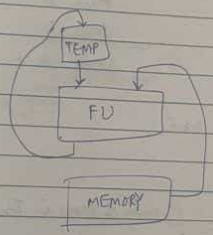
~~Convert to Polish, then~~ (just scan L → R)

*M to write code direct ✓

1-address

oprn	operand
7	32

eg ADD A
SUB B



(Intel)

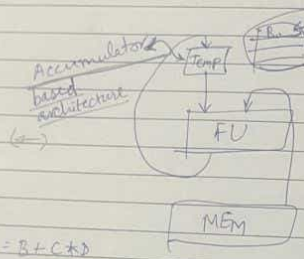
(← was PUSH, POP)

LOAD
STORE

A = B + C
LOAD B
ADD C
STORE A

Lec 15-3-22

ISA



A = B + C * D

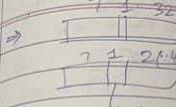
LOAD C
MUL D
ADD B
STORE A

A = B * C + D * E
LOAD B
MUL C
STORE T1 (∵ only 1 space in Temp (accumulator))
LOAD D — in main memory.
MUL E
ADD T1
STORE A

Accumulator is
STILL the bottleneck

↓

e.g. (A) = B + C
D = (A) + E — used multiple times → Store A in some registers
P = (A) + R

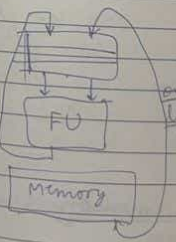


Addressing mode A/M (whether in memory, or registers)

LOAD B	40	7 + 1 + 32
ADD C	40	
STORE R0	10	7 + 1 + 2
LOAD E	40	
ADD R0	10	
STORE D		

→ ∴ no need of making memory reference

e.g. MUL C :
1800 ⇒ get the instr
+ 1000 ⇒ get C
+ 10 ⇒ do the operation
2010 cycles



3 address — | oper | op1 | op2 | dest | = 7 + 2 + 2 + 2 = 13 bit

Load Store : | oper | op1 | op2 | = 7 + 2 + 32
op1 → registers, op2 → memory
In Load, register ← mem, in Store : reg → mem (given addresses)

Page No.
Date

$A = B + C$ (A→R0, B→R1, C→R2)

$D = A + E$ (E→R3)

LOAD R1, B (Load into R1)
LOAD R2, C
ADD R_1, R_2, R_0 7+2×3 = 13
LOAD R3, E
ADD R3, R0, R2
↑ overwritten ... (e.g. ...)
(so performance ↓
if C later used again...)

DEF
→ A = B + C
D = A + E (USE)
P = A + R
S = P + A (USE)
} Liveness Range

Stats : 32 reg. ⇒ 80-85% live var.s hold ✓
64 " ⇒ ~95% " " "
(e.g. 180...)

- Spin Code | e.g. 33rd var.
⇓
Load-Store pair..

* Instructions need to be accessed from memory
e.g. 2010 cycles ⇒ 1000 for accessing the instrⁿ...

$A = B * C + D$ (A→R3, B→R0, C→R1, D→R2)

I_1: MUL R0, R1, R4
I_2: ADD R2, R4, R3

1000 + 10
+ 1000 + 10
2020 cycles

← bring from memory (* slow)

| * making processor very fast : 10 → 1 : not much impact... (∵ 2002 ~ 2020)

⇓

* shud hv some complex instrⁿs... NOT just MUL, etc

MAC : MUL & accumulate.
MAC R_1, R_0, R_2, R_3 : 1000 + 10 + 10 ~ 1020 cycles
↑ $R_0 * R_2 + R_3$ [Lab]

(* 2020 → 1020 by merging instrⁿs ...)

Convolⁿ

$$\sum_{i=0}^{999} A_i B_i = A_0 B_0 + A_1 B_1 + \dots + A_{999} B_{999}$$

By ADD, MUL : 1000 MUL, 999 ADD ~ 2020 × 1000
& by MAC :

By CONV : CONV A, B, C
1000 + (10+10) × 1000 = 1000 + 20,000
= 21000

Insert in linked list

INSERT x, A

Lec 17-3-22

Page No.
Date / /

ISA

Constraints

1. Limited Memory : minimize #bits
2. Slow memory : complex instrⁿ
3. Ease of Programming

```
for ( i=0; i<1000; i++)
    A(i) = B(i) + C(i);
```

A(0) = B(0) + C(0);

⋮

A(999) = .. + .. ;

Can use (ptr.s to addresses) (& store these in registers)

R1 = 5000
R2 = 9000
R3 = 1000

ADD R1, R2, R3 , i.e. add things whose addresses are contents of R_1, R_2 & store at locⁿ with address = content of R3

M[R_3] = M[R_1] + M[R_2]

⇒ Operⁿ | opr1 | opr2 | Dest.

A/M (Addr-ing mode) →
1. Direct Memory
2. Register direct (content = opr.)
3. Register indirect (i.e. address of opr given)
4. Base + Offset (S.T)

Page No. 181

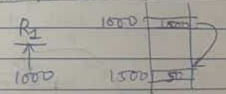
Next iterⁿ :
5004 (∵ 1 operand : 4 bytes)
9004
1004 (∵ *pointers)

A[10] addr. M1 is A + 40 (#bytes) ((R1), 40) offset

4. Base + Offset (~const.)

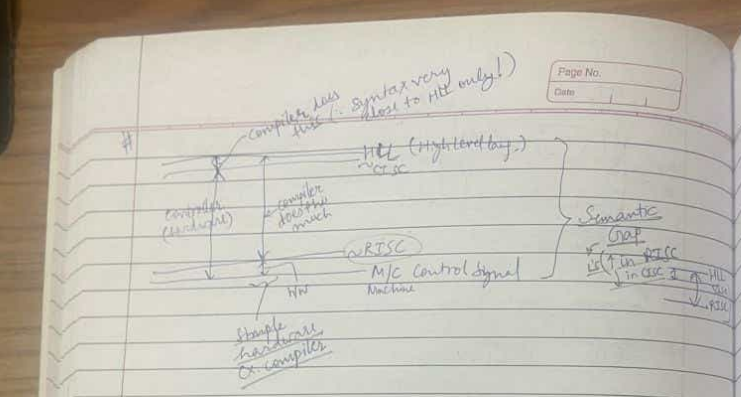
5. Base + Index (2× log(total registers))
i in another register } 2 registers
one R1 (base addr. (A))
other R5 (i)
[R1|R5] ← OPR1 की जगह.

6. Double ptr.
(a/vs:) → MEMORY INDIRECT



7. AUTO INCREMENT / DECREMENT

e.g.
PUSH R1, R2 (contains address of top of stack)
⇕
M(R2) = R1
R2 = R2 + 1 } ← automatically by AUTO INC (ptr. of top of stack incremented..)

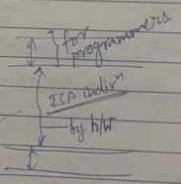


- Only LOAD, STORE can access memory

- ADD A, B, C
  C = A + B — CX. Instⁿ
  LOAD R1, A
  " R2, B
  ADD R1, R2, R3
  STORE R3, C
  (2) — μ-ops. (micro-operᵃˢ)

- Why -O3 not used all the time in gcc?
  ↳ coz time↑ for compilⁿ
  ↑ runtime ↓ (from optimal code)

Instⁿ: Pseudo ISA



μ-operᵃˢ (Micro-operᵃˢ) e.g.↑

AMD: ROPS

Ext translation may use a lookup table.

CISC
RISC

Instⁿ's

Page No.
Date

(Pt. 8 on 182)
One more A/M (for arithmetic/logical instrⁿs).

- IMMEDIATE

6	5	5	(16) here
operⁿ	OPR1	Destⁿ	IMMED (immediate opr2)

I.

i.e. no need to access anything from reg/memory. just these 16 bits are the operand.

To distinguish,
ADD vs ADD I ← Immediate

⇒ now 6 bits ⇒ 32 operⁿs only
↓ but ↓ ⊘

(V.S.)

R type : | 000-000 | | | | (5 bits unused) | FUNC (6 bits) | ⇒ FUNC : 64 possible
000-000 ↓ see from FUNC

I type (Tunnel). | operⁿ | |
= 63 ✓ (∵ 000_000 is for R type)
127 ✓ operⁿs

R
I → 131
J

Memory Commⁿ (DLX..)

- Memory to Reg. : LOAD
  Reg. to Mem. : STORE

① B+D : Base + Displacement.
② B+I.

① (~ I type) | operⁿ | OPR1 (reg.) | DESTⁿ | IMME 16 |

A[10] : R1 → 1000 base ; 8 bytes for floating pt.
IMME 16 ← DISPLACEMENT ; 80 (8×10) — imm. use the value directly
OPR1 : R1 ↑ BASE
DESTⁿ ↓ this reg. has to be loaded.
⇓
1080 locⁿ वाले को R2 में load

LOAD → [$R2 = M(R1+80)$ (Load)

STORE [$M(R1+80) = R2$

LW Word (4 bytes)
SW
LH (half word ⇒ least sig. 16 bits will be loaded)
SH
LB (Byte ⇒ " " 1 byte ")
SB

② Base + Index
(~ R type :) | | | | | R2 |

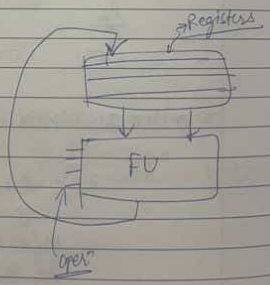
- e.g. i=0, i<1000, i=i+1
constants | M1 is: treat as var., but don't change value
but we've limited memory

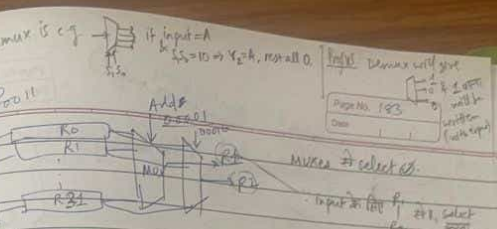
(8) IMMEDIATE (VALUE)
for Compiler
16 bits: ~85% const.s we use are covered
32 " : ~99% " " "

oper^n	A/M	OPR 1	A/M	OPR 2	A/M	DEST^n

e.g. MAC ⇒ 3 OPR.s, 1 Dest^n

7 (5) 5 5
Oper^n OPR1 OPR2 Dest^n
REG REG REG
need to select from the 32 registers
⇒ use MUXes

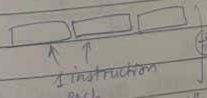




e.g. 0000000 00001 00010 00011
connect to FU
connect to MUXes, DEMUX

- John Cocke (IBM) | ↑ ~ fast bcoz memory accessed
RISC
↳ Reduced Instr^n Set Computer
CISC
↳ Complex/Complete

#- Encoding
oper^n A/M OPR1 A/M OPR2 A/M OPR3 A/M DEST^n (e.g. MAC)
7 = (∵ 8 A/M's) 3
32 (Memory) or 5 (REG)
1 instr^n
3 operands + 1 dest^n
Considering MAX: 7 + (3+32) X4 = 147
140
for fixed len. encoding (RISC)
for CISC: var. len encoding



Can know:
vs
for var len.

Page No.
Date

Lec 21-3-22

ISA

1. Arithmetical logic
2. Memory Communicⁿ (LD, ST)
   ↳ LOAD, STORE.
3. Control Flow Change.
   - Unconditional
   - conditional

CISC
RISC — wt all focus ↑ on RISC
- small instr^s ↓ # A/M^s
- fixed len
- instr^s ↓

cond^al
| oper^n | ADDRESS |

if(i==j)
R1 R2

SUB R_1, R_2, R_3
BEQZ R_3, ADDRESS
(Branch (at equality with zero) — prolly it is : branch if... flag is set.

FLAG
CONDITION CODE REG.
STATUS REGISTER

		overflow, carry..	Z

JZ (Jump on zero).
↳ if flag set ⇒ JZ ADDRESS

(Lab 23-3-22 : NZCO — MSB : flags of CPSR (ARM))

Page No. 187
Date

- J R1

DLX / MIPS → RISC-V
- 32 bits ISA (every instr^n in 4 bytes) (≈RISC)
- 32 Registers : R0 — R31
- Load/store architecture (only these instr^s can access main memory)

- Arithmetic / logical instr^s

R	oper^n	OPR1	OPR2	DEST^N	
31	26	21	16	11	0

32 bits

∵ 32 registers ⇒ 5 bits ✓

(* can access only from reg.f..) — arithmetic, logical

64 oper^s.

AND — ADD R_1, R_2, R_3 = $(R_3 = R_1 + R_2)$
SUB
AND
OR
XOR
SLL : shift left (*2)
SRL : " right (/2)
SLT : set on less than
SLE : " " " or equal to
SGT : " " greater than
SGE : " " " or equal to
SEQ : Set on equal

if $(i < j)$ = [content of R1 is less than j ?]
R1 R2
SLT R1, R2, R3

if $(R_1 < R_2)$ $R_3 = 0 \ldots 0$ (each reg. size is 32 bits)
else $R_3 = 0 \ldots 01$ (not the reverse.)
(if $R_3 = -1$ else $R_3 = 0$)

A() ~ A
{
≡ ← A+4
← A+8
100 → B() → JAL B⊖A ⇒ R31 = 104 (← next instrⁿ)
104 → ≡
}
B() {
≡
500 C() → JAL C → R31 = 504
⇓
104 vanished
504 }
↖ JR R31
⇓
again to 504!!

C()
{
≡
} ← JR R31
⇓
takes to 504.

※ ADD R5, R4, 0 (To move value to another register)
R5 = R4 + 0
≡ R5 = R4

use STACK

← load half word immed. (Net: LH is Load High — MSB 16 bits...)

- LHI Rx, Imm16
e.g. LHI R1, 10H ← Hexadecimal
MSB में ↓
[00 ---- 0] (Reg. is 32 bits long)

- R0 : stores 0 : Read-only (so ADD R0, R0, 5 illegal)
R31 : stores link locⁿ

- ADD (R5), R0, 1000H
→ R5 = [00001000]H ✓ (32 bits ✓)
↓ acting as stack ptr.
→ ~Stack ✓

- SWI R31, (R5)0
(store word immed.) (Base + Displ.)

ADD R5, R5, 4 (Stack ptr..)

⇓

← code में लाने ✓ & then SUB R5, R5, 4
LOAD R31, R5

Example
[00001000]H
Base ↑ A[100] Add all elem.s of A.

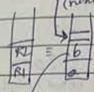
R1 – Counter = 100
R2 – ~~# elements~~ SUM = 0
R3 = 00001000

Initialize : (0)

```
00   ADD R1, R0, 100
04   ADD R2, R0, 0
08   ADD R3, R0, 1000H
Loop: LOAD R4, (R3)0      ← 0 offset
16   ADD R2, R2, R4       ] add to sum
20   ADD R3, R3, 4        ] move to next locⁿ
24   SUB R1, R1, 1
28   BNEQ R1 LOOP         ] if ≠ 0, move to loop. (goto label LOOP)
                          (= -5) [ (20 - 32)/4 = # instⁿs. ]  ← from next one.
32   STORE R2, (R3)0
     HLT
```

~~void~~

```
int main() {
    :
push some vars.  →  fn(a,b)   ] → push these into a stack
pop  →              R1  R2
}
```



R3 (stack ptr) : then (R3)-4, (R3)-8
(next empty locⁿ)

(pushed left to right...) ((can also push R→L))

```
fn( )  ⇒ pop those ⟷  [ R10
{                       [ R11
 ===
}
```

Lec 24-3-22

ISA

- 184 Intel : ISA indirection

- DLX } No program counter (jump...)

→ ARM ISA

- all cond^n^al instr^n^s (?) → document^n^. e.g.

```
        CMP  R0,R1                   CMP sets flags ⊘
        BLT  smaller                       ↑
        MOV  R2,R1        ≡          (CMP R0,R1)
        B    end1        (=)          MOVGE R2,R1
smaller MOV  R2,R0                    MOVLT R2,R0
end1                                    ↑
                                      cond^n^al ⊘..
```

- IF (x==0)
  THEN
    A = A+1
        ↑R1
  |||
  ADDZ R1, R1, #1   (no branch req.)

- R15 is PC (Program counter)   [ARM].
  R15 = R15 + R1
  R15 = R1 + R2 } → i.e. go to addr. R1+R2

- Load, Store multiple registers.

- Simple Instr^n^s
- A few instr^n^ format → R, I, J   (DLX (←))
- A few A/M s
- fixed (len) encoding

here (arm) too ~ ✓

R-type

6	5	5	5	5	6
oper^n^	OPR1	OPR2	Dest^n^		FUNC

00---0 → FUNC ; 5-bit field → undef. (?)

I type (Immed.)

6	5	5	16
oper^n^	OPR1	Dest^n^	IMM16

J-type

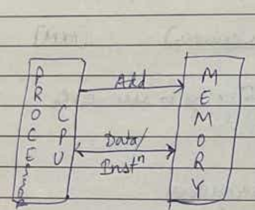
6	26
oper^n^	IMM26

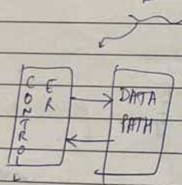
Page No.
Date

① A&L (Arith. & Log.)
- ADD
- SUB
- AND
] let only R-type

② Memory Commⁿ
- LW (Load Word)
- SW
] B+D (Base + Disp) (S-type)

③ Branch
uncondl - J — J-type
condl - BEQ — I-type (e.g. BEQ R_1, R_2, LOC) IMM16





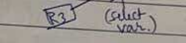
Controller instructs,
Data Path performs (workhorse)
NP

---

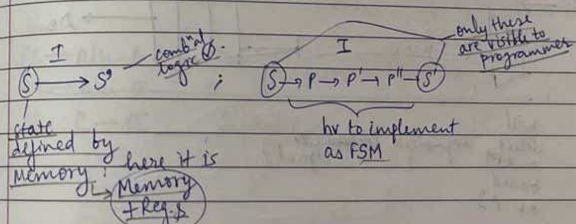
arm isa

DATA PATH
① FⁿAL UNITS → ALU
② STORAGE →
- Programmer's registers (aka architecture reg.)
- Program Counter (PC) / (≡) Instrⁿ ptr [pt-ing to the next instrⁿ to be executed]
- Temporary Storage

③ STEERING LOGIC ↔ Communⁿ infrastr.



CONTROLLER
- Combⁿal logic
- Finite State Machine (FSM)



- Single go NOT possible in CISC. (∵ not even know single instrⁿ or not...) (e.g. we fetch 4 bytes...)
vs
RISC HQ

Control Flow Change. (DLX)

BEQZ, REG, LOC ← I type

oper^n	OPR	DEST^n	IMM16

eg BEQZ R2, 80 ⇔ If R2 is 0
(branch if equal to zero) ⇒ move (80×4 = 320 bytes)
(#instr^{n}s)

* - Instr^n ptr (hv to keep 1 Reg. for this..)
IP = IP + IMM16 × 4
(∵ every instr^n is 4 bytes) (DLX – fixed instr^n size..)

BNEQ REG, LOC

Uncond^{n}al

J: Jump

J LOC

6	26
oper^n	Imm 26

IP = IP + IMM26 × 4
(i.e., this many instr^{n}s away from curr.)

f^n calls: hv to come back to that loc^n

- JAL LOC : ~ store into R31 (→ return addr.)
Jump & Link (R31 gives info of returning addr.)
the curr. addr. from where jumping/leaving

JR Reg (∼R type)

Page No. 191
Date

Lec 22-3-22

DLX

R	oper^n (opcode)	OPR1	OPR2	Dest	Func	
I	oper^n	OPR1	DEST	IMM16		
J	oper^n	IMM26				

Also in pdf DLX – Instr^n set (folder)

Base + Index : R-type
Base + Disp. : I-type

BEQZ Rx, LOC
(equality) relative offset → #instr^{n}s (not #bytes).
(branch)

BNEQ

J (Jump) ⇒ ↑IMM26 ⇒ $2^{26} \times 4 = 2^{28}$ bytes jump possible.
~256 MB (+ fast$_M$: $2^{20} \times 256$)
(2^{10}: 1KB, 2^{20}: 1MB; 2^{30}: 1GB)

JR (jump to register) : JR Rx
register
moves to loc^n given in Rx
(∵ only ~26 bits.. vs 32 bits for 4GB memory)

(fixed for 2)
- R31 contains return addr.
(JAL) LOC → Jump & link.
JALR Rx
Reg
- return can be encoded by JR R31
→ R31 overwritten

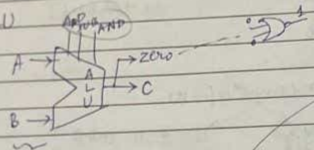
Page No.

Date

- we had $\frac{\#instr^n s}{program} \times CPI \times cycle\ time$

(CPI × cycle time: ~fixed for RISC)

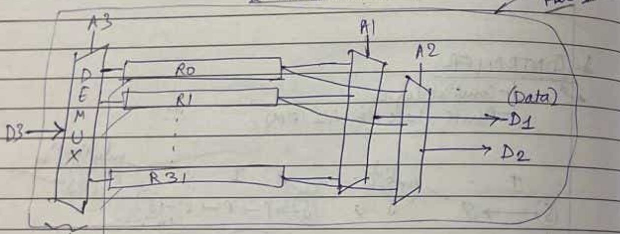
ALU



we hv 32 reg & (ARM too...)
(& each : 32 bits long)

→ can read only 2, & write atmost 1.

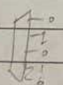
register File (RF)



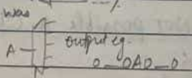
will select one reg. based on A3.

Reg-Write (~ EN signal).

Prof (?): Demux gives



& 1 वाले पर D3 written ☺
(~ contradicts starting के demux ...)
it was



↓ Same as DLX

R-type

6	5	5	5	5	6
oper^n	OPR1	OPR2	Dest^n		FUNC

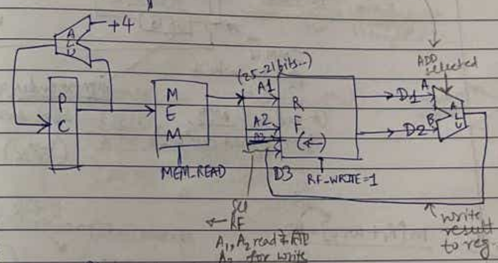
e.g. ADD R1, R2, R3 ($\equiv$ R1 = R2 + R3)

(dest^n last ✓, reorder)

31–26	25–21	20–16	15–11	10–6	5–0
000000	00010	00011	00001		
	R2	R3	R1		ADD

- PC will tell from where to fetch the instr^n. [& update PC].

- PC → Mem. Addr.



and,

PC → alu-A

+4 → alu-B

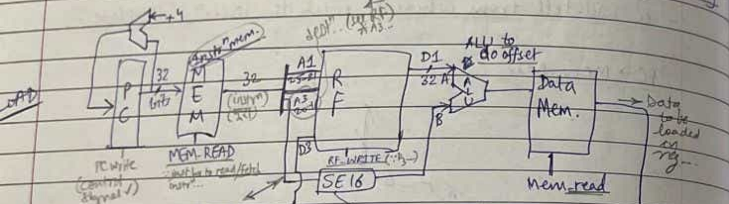
alu-C → PC (PC updated)

Rt pt. Load / store में Imm16×4 नहीं (✗ ~ obv bcoz e.g. R1+39 locⁿ का load करना हुआ तो ?)
vs
BEQ में / J में Imm26×4 ✓ → (~ obv, bcoz there we're branching to another instrⁿ) — multiples of 4 ✓

Memory Commⁿ B+D. I-type

31–26	25–21	20–16	15–0
Operⁿ	OPR1	Destⁿ	IMM16
6	5	5	16
	R1	R2	1000

R2 = M[R1 + 1000] [Load R2 with things at addr R1+1000]

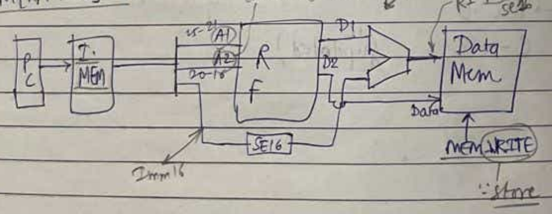


IMM16 : pad to make it 32 bits
(✗ pad with MSB, bcoz signed calcⁿ p...)

Sign Extender
(SEα means α bits को 32 bits...)
(→SE26)

2 mem. reg...

STORE : M[R1+1000] = R2 (stored)

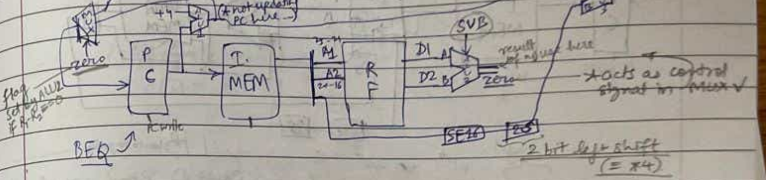


— BEQ R1, R2, 100

31–26	25–21	20–16	15–0
Operⁿ	OPR1	OPR2	IMM16
	R1	R2	100

if $(R_1 == R_2)$ → checked by SUB ↓ : if zero flag set ⇒ ✓.
PC = PC + 4 + (100×4)
(✗ 100 is # instrⁿs... (versus in Load, Store 1000 top: ✗ ~ obv)
else
PC = PC + 4

A1
A2



BEQ ↗

J : PC = PC + 4 + IMM26 × 4 (uncondⁿal)
(J-type)
(SE26)
& no MUX (∵ uncondⁿal, unlike BEQ)
(see NB for "complete"...)
26 bits to 32..

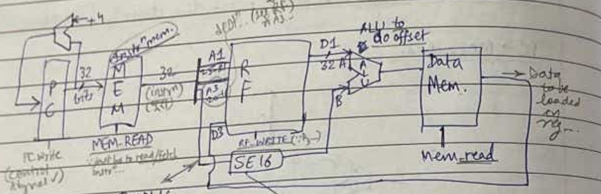
Imp pt: Load, Store मे Imm16×4 नहीं (∗ ~ obv bcoz e.g. R1+39 locⁿ का load करना हो तो!)
vs
BEQ, J मे Imm26×4 (✓ ~ obv, bcoz there we're branching to another instrⁿ) — multiples of 4

Memory Commⁿ B+D. I-type

31–26	25–21	20–16	15–0
Operⁿ	OPR1	Destⁿ	IMM16
6	5	5	16
	R1	R2	1000

R2 = M[R1 + 1000] [Load R2 with things at addr R1+1000]

LOAD

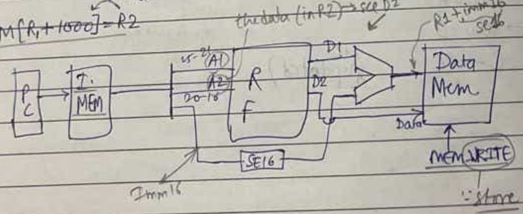


IMM16: pad to make it 32 bits (∗ pad with MSB bcoz signed calcⁿ...)

Sign Extender (SEα means α bits को 32 bits...) (SE26)

∗ 2 Mem. req.

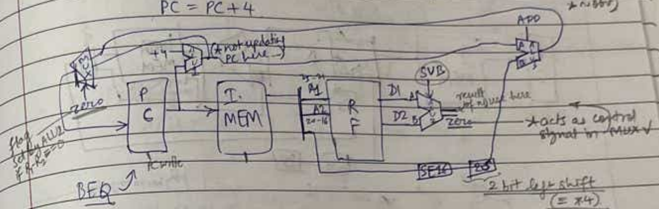
STORE: M[R1+1000] = R2 (stored)



BEQ R1, R2, 100

31–26	25–21	20–16	15–0
Operⁿ	OPR1	OPR2	IMM16
	R1	R2	100

if (R1 == R2) → checked by SUB: if zero flag set → ✓. (A1, A2)
PC = PC + 4 + (100×4)
∗ 100 is # instrⁿ... (vs in Load, Store 1000 top: ∗ addr)
else
PC = PC + 4



BEQ ↗

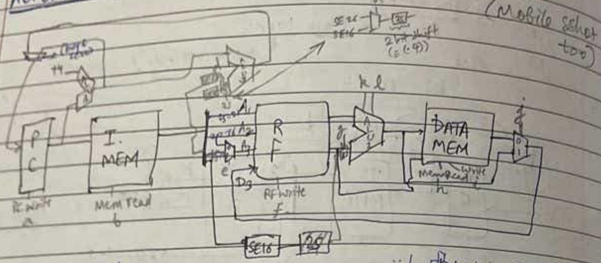
J: PC = PC + 4 + IMM26 × 4 (uncondⁿal)
(J-type)
(SE26) & no MUX (∵ uncondⁿal, unlike BEQ)
(see NB for "complete"...)
26 bits to 32..

Lec 28-3-22

arm

All combined

(Mobile selfie too)



		a	b	c	d	e	f	g	h	i	j	k	l
R	000000	1	1	1 ⇒	x	1	1	0	0	0	1	1	0
L	100100	1	1	1	x	0	1	1	1	0	1		
S	100010	1	1	1	x	0	0	1	0	1	0		
BEQ	001000	1	1	0 (SEN)	x	0	0	0	0	0			
J	010000	1	1	0	1	x	0	X	0	0	0		

(a, b: always 1) (∵ 1 has to go to D_3) (h, i: no data mem in R) (J: uncond^nal)

Fill table & use K-maps on col^n s to get ckt for control vars (comb^nal logic).

- CPI = 1 (every instr^n: 1 cycle). (single go: can see ↑)

$\frac{\text{Time}}{\text{Prog}} = IC \times CPI \times T$
(IC: Dynamic count; CPI: 1)

Mem read / write : 2ns
RF " " : 1 ns
ALU : 2ns

- Longest path: load instr^n

2ns + 1 + 2 (ALU) + 2 + 1 (write RF) (assume MUXes time ≈ 0)
→ 8ns (max)
Freq of oper^n = $\frac{1}{8ns}$ = 125MHz

- for (i=0; i<1000; i++)
A(i) = A(i) + 1

} Static count of instr^n s = 1
Dynamic " " = 1000

↑ways S→S' (∵ single cycle: ✓, e.g. ↑ load..)
Can do S→P→P'→P''→S' too.
↳ Trans^n system.
↳ Sub-tasks
1. Instr^n Fetch
2. Update PC
} 'housekeeping' tasks (have to do)
3. Operand read
4. Instr^n Comput^n or Comput^n of addr. of memory
5. Read/Write Memory
6. Update the state (write to RF).

For A&L Instr
1. Instr^n fetch
2. Update PC
3. Opr. read
4. Instr^n Comput^n (read/write mem: ∅)
5. Write back to RF.

States

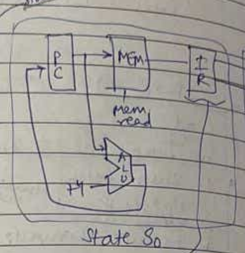
R-type (AL) Arithlog

1. Fetch instrⁿ
2. Update PC
3. Read operand
4. Execute
5. Write back result

State S_0

PC ⟶ Mem. add., alu-A	ADD
Mem. Data ⟶ I. Reg. (Instrⁿ Reg.)	S0
+4 → alu-B	S1
alu C ⟶ PC	

(addr. of instrⁿ to be fetched from mem.) — ALU operⁿ — state ID — next state ID



Instrⁿ in this form

31–26	25–21	20–16	15–11		5–0
OPERⁿ	OPR2	OPR1	Destⁿ		Func
6	5	5	5		6 bits

State S_1

IR_{25-21} ⟶ RF_A_1
IR_{20-16} ⟶ RF_A2
RF-D1 ⟶ T1
RF-D2 ⟶ T2

temp reg.s (can't reuse IR here ∵ use later. operⁿ, destⁿ)

S_2

T1 ⟶ Alu_A	OP
T2 ⟶ alu-B	S2
aluC ⟶ T3	S3

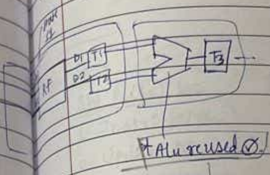
(tho can reuse T1 or T2 ...)
(NOT IR because operⁿ field req...) (see Next state) also (Destⁿ)

S_3

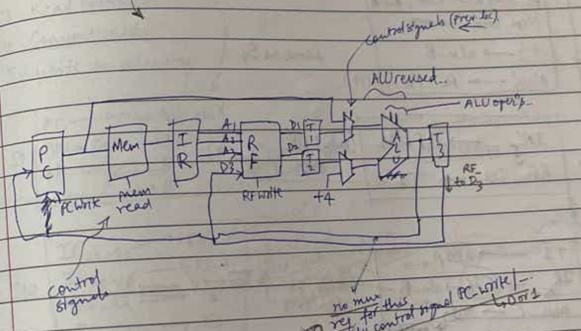
T_3 ⟶ RF_D_3 (to be written...)
IR_{15-11} → RF_A3 (Destⁿ...)

(this is why ... reuse)

$S_0 \longrightarrow S_1 \longrightarrow S_2 \longrightarrow S_3$



ALU reused ⊘.

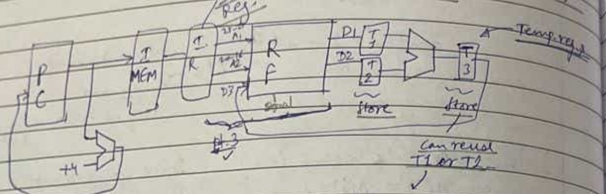


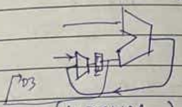
(R-type)

- Can do non-major tasks ||ly (∵ otherwise time ↑..)
  e.g. 1. & 2.
  ↳ ||

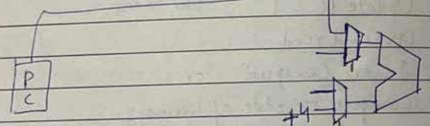
No Instrⁿ

* ∄ need of IR now
* VS: single go में सब हो..
but here, need to store it for next cycle ∅.





Apt: don't actually need this ALU ... bcoz at this state/cycle the later ALU is free..



FSM ↑ : 1st clock tick ... job
2nd " " ... job
⋮

Lec 29-3-22

Page No. 265

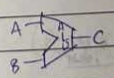
- $S \to S'$
  CPI = 1
  T = 8ns
  $\frac{Time}{Prog} = IC \times CPI \times T$ (← time/cycle)
  ... × 1 × 8ns
  MIPS = 125 ✓ (∵ 125 MHz..) → $\frac{1}{1 \times 8 \times 10^{-9}}$
  ↳ $CPI \times \frac{Time}{Cycle} = \frac{Time}{Instr^n}$

- $S \to P \to P' \to S'$
  Load : 8ns
  But, e.g. Jump : 4ns ⇒ 4ns idle ⇒ break down as ∅
  1. Instrⁿ fetch } housekeeping tasks — can do concurrently, rest sequentially..
  2. Update PC }
  3. Operand Read
  4. Execute / Compute the addr.
  5. Mem. access
  6. Write back result

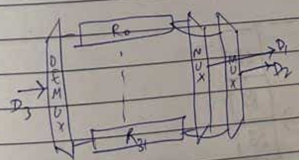
Load = 8ns
AL (Arith, Log.) = 6ns
(Store) ST = 7ns
Branch = 5ns
J = 4ns
(?) Values (vs later 214..)

Data Path

PC



- Register file



- Point-to-point.

Load ~~EA~~ B+D (Base + Displ.)

LW R_1, (R_2), 100 : R2 = M[R1 + Imm16] → reg. into which value is loaded

I-type

31–26	25–21	20–16	15–0
operⁿ	OPR1	Destⁿ	IMM 16
6	5	5	16

1. Fetch instrⁿ
2. Update PC } ‖ly ⊙ in S4 ✓
3. Read operand
4. Compute addr.
5. Read memory
6. Write back result into RF

(3–6: "Load")

S4
- PC → mem.addr., alu-A
- Mem. Data → IR
- +4 → alu-B
- aluC → PC (PC-write).

Same as S_0

S5
- IR_{25-21} → RF-A1 (Only 1 OPR here.) ("base")
- RF_D1 → T1

S6
- T1 → alu A
- IR_{15-0} → (SE16) → alu-B → *No "4" (*200 top ✓ ~dir) (No "25")
- alu_C → T3 → contains the mem locⁿ..

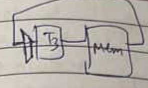
*↓ pt : no need of 2 separate memories now (vs single go ऑफ़.)

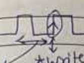
S7
- T3 → Mem-Ad
- Mem. Data → T3 → has the load result ✓ (Write into RF) ("loaded" ✓)

S8
- T3 → RF-D3 (demux opt.)
- IR_{20-16} → RF-A3 (loaded into register)

Destⁿ ; demux control... ← 198

$S_4 \rightarrow S_5 \rightarrow S_6 \rightarrow S_7 \rightarrow S_8$ for Load operⁿ.





read ✓ *write operⁿ occurs at end of cycle (at this rising edge).

Store

SW R_1(R_2), 100

31–26	25–21	20–16	15–0
Operⁿ	OPR1	Destⁿ	Imm 16

1. Instrⁿ fetch
2. Update PC
3. Read operands
4. Compute addr.
5. Write to memory

S9
- PC → Mem.Addr, alu A
- Mem Data → IR
- +4 → Alu-B
- alu_C → PC

$S_9 \rightarrow S_{10} \rightarrow S_{11} \rightarrow S_{12}$ 4 steps/states.

S10
- IR_{25-21} → RFA1 ("base")
- IR_{20-16} → RF-A2 *reading of destⁿ done in advance (→ this has the value to be stored.) reg ↔ $M[R_1 + Imm16] = R2$ (R2's contents stored...)
- RF_D1 → T1
- RF_D2 → T2

S11
- T1 → alu A
- IR_{15-0} → SE16 → aluB
- aluC → T3

T3 now has R_1 + Imm16 (mem. addr.)

S12
- T3 → ~~Mem~~ Addr.
- T2 → Mem.D → Stored ✓

Lec. 31-3-22

Branch: BEQ R_{s1}, R_{s2}, Imm

I-type	Operⁿ	OPR 1	Destⁿ	IMM 16
	31 – 26	25 – 21	20 – 16 (~OPR2)	15 – 0

if (R_{s1} == R_{s2}) then
  PC = PC + 4 + Imm16 × 4
  * ∵ Imm16 is #instrⁿs (BEQ, J) (* 200 top : ~ (?))

Else
  PC = PC + 4

1. Fetch instrⁿ } "housekeeping" tasks
2. Update PC
3. Read operands (R_{s1} & R_{s2})
4. Compare (subtraction)
5. If zero then PC = PC + 4 + Imm16 × 4

A, B → ALU → C, zero (flag set if zero result)

S13:
- PC → mem_Addr, alu_A
- +4 → alu_B
- alu_C → PC
- Mem. Data → IR (i.e. instrⁿ fetched from memory, put into IR)

↓

S14:
- IR_{25-21} → RF_A1
- IR_{20-16} → RF_A2
- RF_D1 → T1 (temp. reg.)
- RF_D2 → T2
- PC → alu_A
- IR_{15-0} → SE_{16} → 2S → alu_B
- alu_C → T3

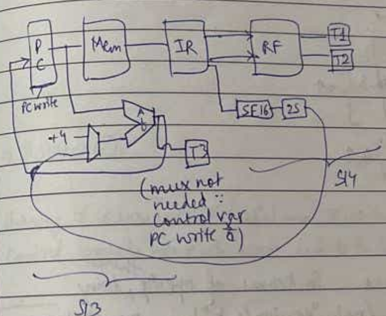
~ loaded in S13...

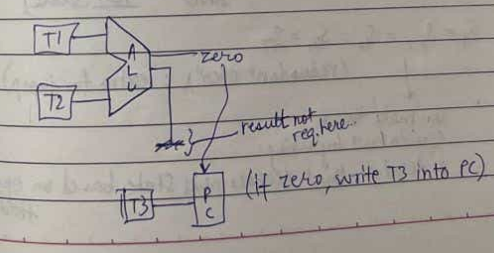
S15:
- T1 → alu_A } for subtrⁿ
- T2 → alu_B
- PC → alu2_A
- IR_{15-0} → SE16 → 2S → alu2_B
- if (zero == 1) then
  alu2_C → PC
  → T3

4 pt. 4, 5 can do ~ ∥ bcoz PC+4 +Imm16×4 computed first, then if zero → load into PC (otherwise nothing) — just PC ← PC+4

Aim: minimize #states so that CPI ↓.
& here: 2 ALUs req...
BUT can do alu2 tasks in S14 itself !!
Optimized in pencil. (PC pre-computed)

$S_{13} \rightarrow S_{14} \rightarrow S_{15}$. & only 1 alu
& " " memory } vs in single cycle 2 mems & 3 ALUs.





Jump instrⁿ : J-type format.

operⁿ	IMM26

1. Fetch instrⁿ
2. Update PC ← PC+4
3. Compute PC+4+Imm26×4
4. Update PC

(no "Read OPR" here, ofc)

(∵ when fetching instrⁿ, don't know which instrⁿ it is... ∴ ‖ they're ∴.)

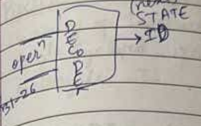
S16
- PC → Mem Addr, alu-A
- +4 → alu-B
- alu-C → PC
- Mem. Data → IR

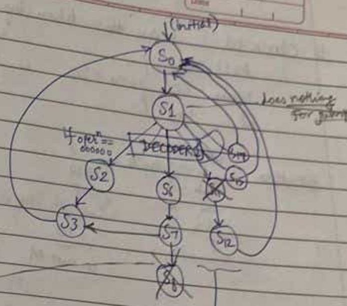
↓

S17
- PC → alu-A
- IR_{25-0} → SE26 (sign extend 26 bit) → 32 bit → 2S → alu-B } 3.↑
- alu-C → PC } 4.↑

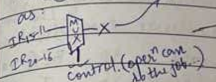
S0, S_4, S_9, S13, S_{16} same: in terms of operⁿs they're doing.
(not "equivalent", bcoz next states not same).

Can make $S_1 = S_5 = S_{10} = S_{14} = S_{17}$
(redundant operⁿs ; extra for jump)
↓
Can make these equivalent by using a decoder, that decodes next state based on operⁿ field.

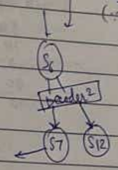




Can combine S_3, S_8 ✓ tho $IR_{(\ )}$ → diff : IR_X
as:



Can combine S_6, S_{11} if we use another decoder... (∵ NOT equivalent, but operⁿs same.)



⇒ 9 states ∅.

- Sharing of resources ✓ (add'al mem, ALU ⊗, but regs T₁... ∅)
- Control signals for each state need to be decided.

(↑ ~ assembly line)

	IF	ID	EX	MEM	WB
cycle 1:	I_1				
2	I_2	I_1			
3	I_3	I_2	I_1		
4	I_4	I_3	I_2	I_1	
5	I_5	4	3	2	1
6	6	5	4	3	2

LATENCY

(assuming independent instr^{n}s & not data-dep... (no control flows..)

after 5 cycles, 1 instr^n out every cycle ; so, looks like every instr^n in 1 cycle !!

$\Rightarrow$ CPI = 1

& T also ↓ because all "stages" doing work for different instructions

$\Rightarrow$ ~ multicycle ↓ T

$\Rightarrow$ T = 2ns.

$\Rightarrow$ IC × CPI × T ~ 4 times improvement

Hw to detect..

① Data Dependency

I_1 : $r_1 = r_2 + r_3$

I_2 : $r_4 = r_1 + r_5$ dependency

r_1 10
r_2 20
r_3 30
r_4 40
r_5 50

Characteristics of pipeline :

1. fetches 1 instr^n at a time
2. Unified pipelines
3. Lock step advancement (can't advance until all advances)

	IF	ID	EX	MEM	WB
I_3	(I_2)	I_1			

gets I_1 = 50 ← 20+30 = 50 computed & stored in pipeline register (EX/MEM)

if dest^n of I_1 = src of I_2, then problem

↳ if R^{ID}_{s1} == R^{EX}_{d} or R^{ID}_{s2} == R^{EX}_{d} (source 1, 2)

if so, do :

I_3	I_2	I_1		
I_3	I_2	–	I_1	
I_3	I_2	–	–	I_1 (written back)
I_3	I_2	–	–	–
I_4	I_3	I_2	–	–

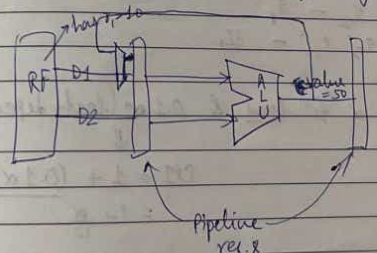
3 cycles ~ wasted on such dependency. (∵ 3 cycles : movement ↓↑ ⊗)

eg. 10% dep ⇒ $1 \times 0.9 + 3 \times 0.1 = 1.2$ (CPI)

penalty for 10%

⇒ Can do : read from pipelined reg. ("50" EX/MEM)

↳ k/a Data fwd-ing / Data bypassing.



Page No.
Date

Clock pd : max. time taken by any state

ALU : 2ns
Mem : 2ns } max = 2ns vs earlier : 8ns
RF : 1ns

~~section~~ : we have

R →	4 cycles	— 8
LW	5	— 10
SW	4	— 8
BEQ	3	— 6
J	3	— 6 ns

in curr. M.

Time = IC × CPI × t (→ 2ns)

CPI ↓ depends on freq. of operᵍ.

e.g.

R	50%
LW	20%
SW	10%
BEQ	10%
J	10%

⇓

CPI = 4 × 0.5 + ... + 3 × 0.1 (wgted mean) = 4

⇓

Time same as single cycle!! (IC × 4×2ns = 8ns = 1×8ns)

But, if 50%, 10, 10, 20, 10 → (3.8) CPI ⇓ Performance better

- Single cycle : CPI = 1 (best)
- multi " : t = 2ns (")

Assume
- all instrⁿs are indep. of each other
- no change in control flow (i.e. top-bottom execⁿ ✓), then CPI = 1, t = 2ns ✓ — pipeline archi

(bridge, car assembly, bricks ↑ M)

Page No. 215
Date

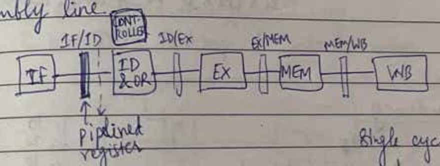
Lec 3-4-22.

Pipelined Architecture
- Chaining of Instructions
- Single Cycle : CPI = 1
- Multicycle. T↓

Subtasks
1. Instrⁿ fetch (IF)
2. Operand read (ID : Instrⁿ decode ✓)
3. Execute (EX).
4. Memory Access (Mem)
5. Write back result (WB)

	IF	ID	Ex	Mem	WB
AL ·	IF —	ID —	EX —	~~WB~~ —	WB
L	IF —	ID —	EX —	Mem —	WB
S	"	"	"	"	—
B	"	"	"	—	—
J	"	"	"	—	—

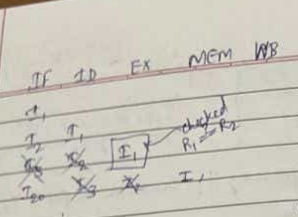
Assembly line



Single cycle
↓
Controller (for control signals)
(Combⁿal logic ; e.g. K-maps ...)

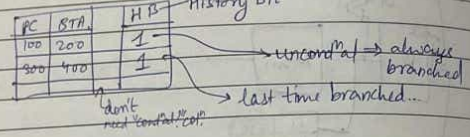
NP →

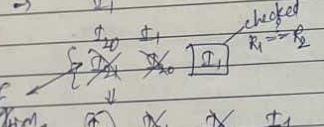
IF ID EX MEM WB



} 2 cycles wasted if $R_1 == R_2$

- Can predict: do: If last time branch taken (i.e. branched), then branch this time too

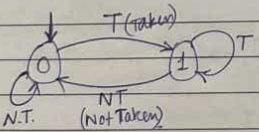




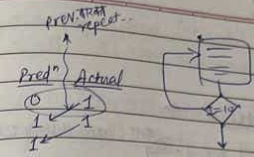
if wrong prediction

to fetch this, hv to wait PC+4,
but we already changed it to PC+4+(Imm16×4) for I_{20}...
⇒ use this.

Branch / Smith Predictor



HB is the "memory" for this FSM (i.e. state label)



⇒ 2 wrong decisions (while entering, exiting)

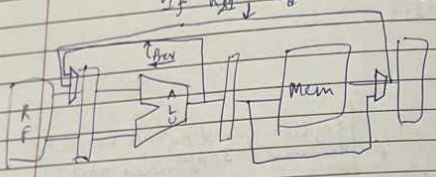
to ↑ efficiency, need to look at more earlier history.

Page No.
Date

Dependency ⟲ 2

I_1 : $r_4 = r_2 + r_3$
I_2 : $r_4 = r_5 + r_4$
I_3 : $r_4 = r_4 + r_5$

If $R_{dd}^{ID} == R_s^{Mem}$ or $R_{dc}^{ID} == R_d^{Mem}$



I_1 : $r_4 \leftarrow Mem(r_4 + r_5)$ (i.e. M[ ]) (Load)
I_2 = $r_4 + r_3$

1
2
3
4 I_3 I_2 – I_1
5 I_4 I_3 I_2 – I_1

If α fn load & $0.1 \propto$ (load + dependency)

$$CPI = 1 + (0.1\alpha) \times 1$$
$$= 1 + \frac{\alpha}{10}$$

Page No. 2.19
Date

~~Compiler can help~~:

eg $a = b + c$
$d = a + e$

Load r_1, b
Load r_2, c
Add r_3, r_1, r_2
Store r_3, a
Load r_4, e
Add r_5, r_4, r_3
Store r_5, d

2 dep

Try to remove these immediate dependencies.

Shift it. ⊗

Load r_4, e

now this dep does not cause stopping (can use data bypassing)

a/k/a Code Reordering

Lec 5-4-22

Pipelined Architecture Contd.

216 Chor A
Assumpt^n s
① Independence of Instr's
② No change in control flow

I_1 : J loc
I_2 : ADD R_1, R_2, R_3
⋮
$I_{20}^{(loc)}$: SUB R_4, R_5, R_6

	IF	ID	EX	MEM	WB
1	I_1				
2	~~I_2~~	I_1 (detected change ✓)			
3	~~I_3~~	~~I_2~~	I_1 (comput^n of Addr. (I_{20}))		
4	~~I_4~~ I_{20}	I_3	I_2	I_1	
5	I_5	I_4	I_3	I_2	I_1
6	I_6	I_5	I_4	I_3	I_2

latency

4	I_{20}	~~I_3~~	~~I_2~~	I_1	
5	I_{21}	I_{20}	~~I_3~~	~~I_2~~	I_1
6	I_{22}	21	20	~~X~~	~~X~~
7	23	22	21	20	~~X~~

don't Write back.
2 cycles ~ waste : no output (for every uncond^nal B)

effect on CPI:
α - uncond^nal

20% branches
↳ 40% uncond^nal B.

$CPI = 1 + 0.2 \times 0.4 \times (2)$
$= 1.16$

100 I_1
⋮
200 I_{20}

Page No. 231

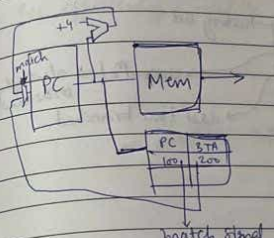
Can do:

PC	BTA
100	200

BTA: Branch Target Addr.
fill table after 1st time exec^n of I_1... (& assuming I_1 exec. many times)

⇓
directly after fetching I_1, can know to fetch I_{20} ✓

I_1
I_{20} I_1 ...
I_{21} I_{20} I_1
⋮
no CPI ↑se



match signal (for MUX)

Cond^nal Branches:

(300) I_1 : BEQ R_1, R_2, loc
I_2 : ADD R4, R5, R6
⋮
(400) I_{20} : loc SUB R_6, R_7, R_8

if $R_1 \neq R_2$ then good ✓ I_3 I_2 I_1 ...
else, I_3 I_2 useless.
NP
↓

- if no indep. instr^n found → put NOP (no oper^n) after branch
- DLX / MIPS / SPARC : h/w Branch Delay Slot.

So, if $r_1 = r_0 + r_3$, beq r_4, r_5, skip → shuffle

⇓

to compute 'skip' addr, do PC + 4 + IMM×4

(PC + 4 ← the "next" addr.)

∵ programmer coded wrt $r_1 = \ldots$, beq…

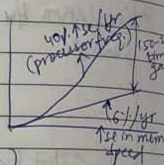
but addr.p wrt beq…, $r_4 = \ldots$

IF (Instr^n Mem) — ID — EX — MEM (Data Mem.) — WB

very slow (200 cycles gone till mem. responds)

⇓

CPI = 200



$$\frac{\text{Data}}{\text{Second}} = \frac{\text{Data}}{\text{Instr}^n} \times \frac{\text{Instr}^n\text{'s}}{\text{Cycle}} \times \frac{\text{Cycle}}{\text{Second}}$$

- BW (Bandwidth)
- Data/Instr^n: 4B (32 bits) → in case 20% load/store → (4 + 0.2×4)B
- Instr^{n}s/Cycle: 1 = $(CPI)^{-1}$ ← max IPC
- Cycle/Second: 10^9

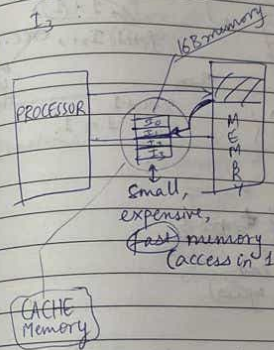
= 12 GBps = 96 Gbps

⇒ 4.8×3 GBps

e.g. Core i7 : 4 cores, each takes 16 B
↓
4.8 × 3 × 16 Gbps } this kind of BW needed.

```
# for (i=0; i<1000; i++) {
      I0 :
      I1 :
      I2 :
      I3
  }
```

(locality of reference)



200 + 1 (other steps (think ~ assembly line); time overlapped by)
1 + 1
1 + 1
1 + 1
⇓
204

Next time
1 + 1
1 + 1
1 + 1
1 + 1 (overlapped by next…)
4

⇓

204 + 4×999

otherwise: 200×4×1000 ($I_0 \ldots I_3$)

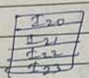
⇓

200 × 4 × 1000 / 4200

CPI: $\frac{4200}{4000}$ = 1.05

# for (i=0; i<1000; i++) {
  if (i%2 == 0) {
    I_{20} ;
    I_{21} ;
  }
  else {
    I_0 ;
    I_1 ;
  }

→ 1st iterⁿ



200 + (1) — exec. I_{20}.
1 + (1) — fetch I_{21}, exec. it — overlapped under the next 200

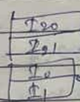
2nd iterⁿ { then
200+1
1 + 1

⇓

$CPI = \frac{201}{2} = 100.5$

(2 instrⁿs exec. (in 201 cycles))

→ do ↓ : separate chunks. (8 Bytes each) here



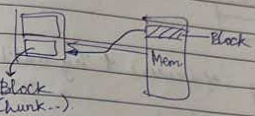
cycles: 1st iterⁿ : 200+1, 1+1 } 201
2nd " : 200+1, 1+1 } 201
3rd " : 1+1, 1+1 } 2
⋮ } 2
⋮

$CPI = \frac{201 + 201 + 2 + \ldots + 2}{2000}$ (# instrⁿs) $= \frac{402 + 2\times998}{2000}$

$= \frac{398 + 4 + 2\times998}{2000}$

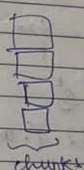
$= \frac{2398}{2000} \sim 1.2$ (close to 1)

⇒ bring instrⁿs b/w 2 branches



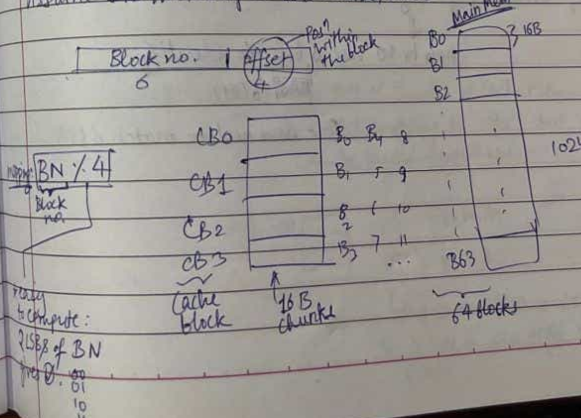
32kB cache mem.

Questions ↑ :



1. Where to place?
2. How to identify? — b/c I_{21} etc. & memory address
3. Whom to evict?
4. How to handle write? (∵ data in cache, memory)

Assume 1024B memory & 16B blocks



these bits give CB#

To identify e.g. B_1, B_5, B_9 (i.e. same CB#)
⇓
maintain TAG bits
(MSB of [6 | 4])

TAG
D E C O D E R → 0000 → B1, 0001 → B5, ... (MSB 4 bits)

Indexing delay (of decoder) + Tag match delay +.

2 bits only (∵ 4 chunks)

but

e.g. $I_0 - I_1$ → B_1; $I_{20} - I_{21}$ → B_9 (example)

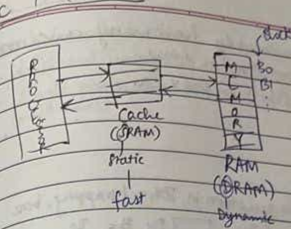
~~same chunk~~ ← → performance ↓↓. (∵ will kick out the other). 1 iter, bring B_1, next, bring B_9...

So, allow to go anywhere in the 4 chunks
⇓
now hv to use 6 bits to identify
[TAG] (6 bits).

But now need to match 6 bits...

Lec 7-4-22

Page No. 231



4 questions 229

① M1 (set") Direct Mapping

Problem: e.g. for (____)
if (i%2 == 0)
B1 — {I_0; ...
else I_{20}; ...
{B9

then after every iter CB1 content replaced.

M2: e.g. 4GB RAM, 64B Blocks, 32 KB cache → 512 things to compare in ||.
"Fully Associative"

⇓

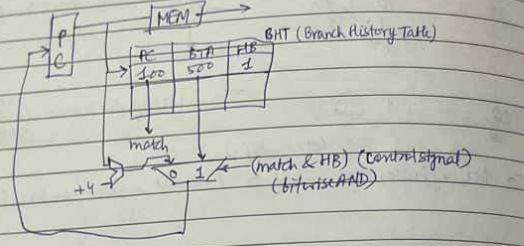
M3: Middle Way: make sets of chunks of Cache Mem.
& direct mapping to sets, but inside set, can be placed anywhere.

SET ASSOCIATIVE MAPPING (most common used)

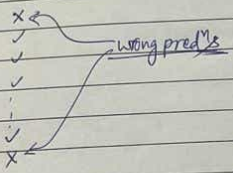
e.g. CB 0, 1, 2, 3 → S0 {B0, B2, B4, ...; S1 {B1, B3, B5, ...
↳ 1 bit req. (LSB = %2)
& 5 bits to identify within set.
(TAG)

Lec 6-4-22

Branch Predictor

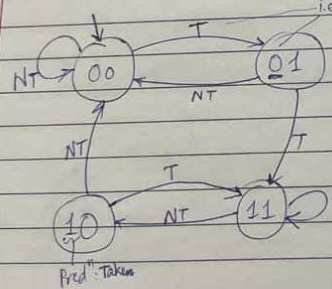


e.g. of loop



⇒ use 2 Bits (History)

00 ↑ use MSB for pred" (0: NT, 1: T) — not taken, taken



⇒ Now only 1 wrong decision (at exit)
(1st time of all: 3 wrong decisions — then only 1 wrong per loop...)

2 bits: 3 + 999×1 out of 1000×1000 → 85-90% accuracy
Single bit: ~80-85%.

Example: α [Ir" / B], 90% accuracy
↳ Ø pred" → no loss in → I_1 I_2 I_3
Ø " → 2 cycles' loss

$CPI = 1 + \alpha \times \frac{1}{10} \times 2$
$= (1 + 0.2\alpha)$ if $\alpha = 0.2$ → $1 + 0.04$
$= 1.04$

Correlation b/w branches

Example:
if (i%2 == 0) then... → if NT, then NT Ø (pred")
if (i == j) then...
if (i%10 == 0)...

[e.g. BI MOD predictor 2 level]

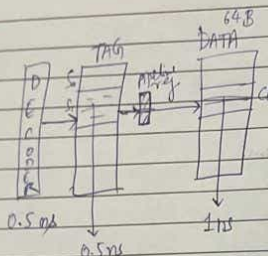
(↙ 2bit branch predictor considering all branches as independent)

I_1: $r_1 = r_2 + r_3$
I_2: beq r_4, r_5, skip
I_3: $r_7 = r_8 + r_9$

skip: $r_3 = r_4 + r_7$
I_{20}

Can put a rule in ISA: an instr" is always executed after branch.
↓
can do: reshuffle I_2 ... I_1 ...
must be indep of branch
↓
1 cycle saved

Even if $h = 100\%$ (hit rate), then CPI ~ 1.15
data dep., control flow



$t_{dec} + t_{tag} + t_{data}$
2 ns
2 cycles for 1 GHz

Pipelined Caches

IF1 — IF2 — ID — EX — MEM1 — MEM2 — WB

$IPC = \frac{1}{CPI}$
Instr's per cycle

CPI ≈ 1 (1.3/1.4) for pipelined

* can pick instr's from multiple src.s ⇒ ~independent

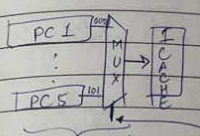
	IF	ID	EX	MEM	WB
src of this instr	I_1^1				
	I_1^2	I_1^1			
	I_1^3	I_1^2	I_1^1		
	I_1^4	I_1^3	I_1^2	I_1^1	
	I_1^5	I_1^4	I_1^3	I_1^2	I_1^1
	I_2^1	.	.	.	~out

all src.s diff. (+80 control flow, branching etc.) ⇒ no problem).
- No data dep.

✓ CPI = 5 for each src.
But combined, CPI = 1

Context (each process has a)
1. Memory (Core segment / Data segment / Stack segment)
2. RF
3. PC
4. Files

Need multiple PC's

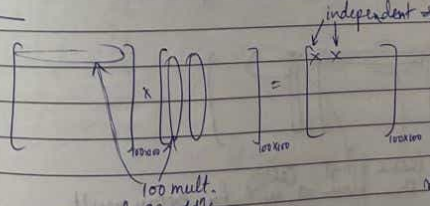


for fetching from different sources.

modulo 5 counter will work ✓ (FSM)

& multiple RF's

e.g.



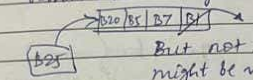
independent ⇒ can do concurrently
⇓
make multiple threads & then run them concur. as.
⇓
need diff. SS, RF, PC

#Blocks in a set : determined by how many concurrent matches can be done...
e.g. 5 bits in prev ex.

Qs 1, 2 done

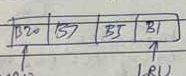
Q3. Whom to evict? [no question in Direct Mapping, bcoz e.g. [ ] B1 B5 B9... ⇒ kick out existing.]

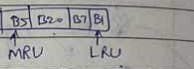
- can do: remove blocks by FIFO

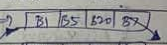


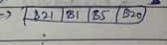
But not an efficient M, bcoz B1 might be used...

- LRU: Least recently used

e.g. 

Now B5 used ⇒ do 

then B1 ⇒ 

then B21 ⇒ 

then :

hv to maintain queues for each set (for set associative mapping)

hv to do lot of work

⇓

approximate : FIFO BUT NOT MRU

Q4. How to handle writes:
- ~~different~~ images in cache & memory
- single processor ⇒ first see if not in cache. If hit ⇒ use that. Else miss ⇒ use from memory
(multi-how?)

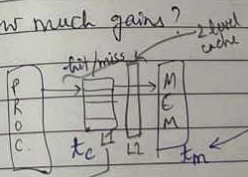
↳ to signify that a CB is modified ⇒ maintain 1 bit

- if modified, then if evicting, write back to memory...
↳ this can use a temp. loc^n & then write back from it (to save time)

Write-back policy

- Write-Through policy
↳ while modifying cache ⇒ write into memory too
(✓ also for multiple processors). ↑ but memory traffic ↑...
↳ while evicting, no dirty bit req....

How much gains?



hit rate : h (~90-95%)
miss " : $m = 1-h$

$t_A = t_c + m \cdot t_m$ (? regarding time?)

∵ always cache ① accessed ① first...

$= 1 + 0.1 \times 200 = 21$

Page No.
Date

for 2 level cache

$t_A = t_{L1} + m_1 t_{L2} + m_1 m_2 t_M$

$= 1 + 0.1 \times 10 + 0.01 \times 200$

$= \boxed{4}$

$(h_1 = h_2 = 0.9)$

10 cycles to access L2 cache

- 3-level cache

32 KB	512 KB	2MB
1	10	50

$\Downarrow \quad (h_i = 90\%) \; i = 1,2,3$

$CPI = t_{L1} + m_1 t_{L2} + m_1 m_2 t_{L3} + m_1 m_2 m_3 t_m$

$= 1 + 0.1 \times 10 + 0.1 \times 0.1 \times 50 + (0.1)^3 \times 200$

$= 1 + 1 + 0.5 + 0.2$

$= \boxed{2.7}$

(diminishing returns!)

Page No. 235
Date

Single cycle

Sec 11-4-21

1. Hierarchy of Memory

L1 – L2 – L3 – Main Mem.

32 KB | 256 KB | 2MB | 16GB

1-2 cycles | 10 cycles | 40 cycles | 150 cycles

$t_A = t_{L_1} + m_1 t_{L_2} + m_1 m_2 t_{L_3} + m_1 m_2 m_3 t_m$

3C-model

1. Compulsory Miss (cold start) (~ first time brought the block)
2. Capacity miss
3. Conflict miss — max in the direct mapping, min in fully assoc.

further capacity miss↑ (cannot afford large size cache)

IF ID EX MEM WB

(* Single cycle, Pipelined ✓ 2 mem. ✓)

* all in ∥ (on different I's)

P C → I. Mem

L1-I cache 32 KB

D. MEM

L1-D cache 32 KB

Tue. 12-4-22

Multiple RFs for : e.g. $R_1' = R_2' + R_3'$
and $R_4^2 = R_2^2 + R_3$

To improve CPI of individual srch, utilize
e.g. matrices.

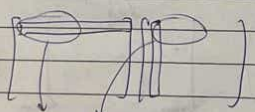
- Threads (theoretically, 10^2 ✓ for K) :: indep.
  - execute same instr's : 100 mult + 99 add's
  - ~~private~~ PC, ~~RAF~~, ~~STACK~~, Archi. RF
  - SIMD (for IF ID EX MEM WB)
  - gives CPI ~ 1 for the matrix ex.

Parallel Programming

e.g. for (i=0; i<1000; i++) → pthread
✓ A(i) = B(i) + C(i)
✗ A(i) = A(i-1) + B(i) → threads not indep.
(dependency)

- Logical cores (my: 2× Physical cores)
  - can provide indep. stream of instr's.

- Example:

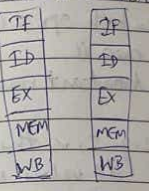


Cache brings this block ⇒ cache miss for every mult. (2nd matrix)

- so A·B ⇒ store B as B^T to prevent this.

- Graphs : CSR/CSC

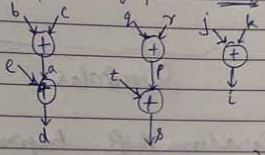
- To use CPI < 1 : Fetch multiple instr's at once (use multiple pipes)



✓ If indep. instr's
BUT ✗

- I_1 a = b + c
  I_2 d = a + e
  I_3 p = q + r
  I_4 s = p + t
  I_5 i = j + k

Data flow / Control graphs: DFG



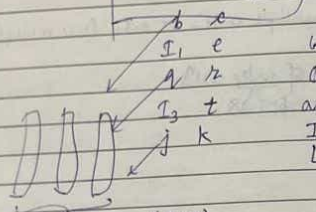
Theoretically, I_1, I_3, I_5 in ∥ → "1 cycle" (just theo.) → 2 cycles
Then I_2, I_4 " "

⇒ I_1
I_3
I_5
I_2
I_4

IPC = 5/2 = 2.5 (>1!)

To produce the DFG

Temp. Storage
e.g. d=@1 e
↳ tag, that I_1 will produce this



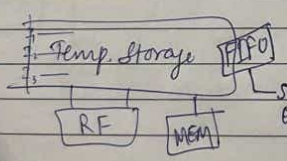
f'al pipes (II∅)

When both opr. & ✓ for any I_i, can exec. it, and broadcast result with tag I_i.
↳ e.g. I_1 ✓, then broadcast d's value with tag I_1,
⇓
I_2 too ready to be exec. ✓

Superscalar

- Paradigm shift: Program ~~exec~~ guided exec^n (PC,..)
to
Data guided exec^n.

- Issues :- debugging difficult. (∵ e.g. just a stmt hv been updated, but in 1 cycle, a, p, i updated)
⇓
- use temporary storage (e.g. don't write p, i into RFs (so that programmer can debug), but store in temp ✓)



so program order ✓

- SMT, P4 (Hyperthreading)
- can use different instr^n streams

Size, Associativity of caches.