Single Cycle :

$$T_c = t_{pcq\_PC} + 2t_{mem} + t_{RFread} + t_{ALU} + t_{mux} + t_{RFwrite}$$

CPI = 1 → The cycles per instruction decide by LW

Throughput = 1 = $\frac{1}{CPI}$

ALUOp	Meaning
00	add
01	subtract
10	Look at *funct* field
11	n/a

Instr.	Jump	RegDst	RegWrite	ALUSrc	Branch	ALUOp1	ALUOp0	MemRead	MemWrite	MemtoReg
R-type	0	1	1	0	0	1	0	0	0	0
lw	0	0	1	1	0	0	0	1	0	1
sw	0	x	0	1	0	0	0	0	1	x
addi	0	0	1	1	0	0	0	0	0	0
B-type	0	x	0	0	1	0	1	0	0	x
J-type	1	x	0	x	x	x	x	0	0	x

Multicycle :

Fetch

Machine state	Operation	Control signals
T0	InsR ← M[PC]; PC ←PC+4	IorD=0, IRWrite=1, ALUSrcA=0, ALUSrc=01, ALUOp=00, PCSrc=0, PCWrite=1

Decode

Machine state	Operation	Control signals
T1	(PC+4) + SigExtn(offset)	ALUSrcA=0, $ALUSrcB_{1:0}$ = 11, ALUOp=00

LW

Machine state	Operation	Control signals
T2	A + sigEx(offset)	ALUSrcA=1, $ALUSrcB_{1:0}$ = 10, ALUOp=00
T3	Data ←M[A+sigEx(off)]	IorD=1
T4	RF[dest] ←Data	RegDst=0, MemtoReg=1, RegWrite=1, T0

SW

Machine state	Operation	Control signals
T2	A + sigEx(offset)	ALUSrcA=1, $ALUSrcB_{1:0}$ = 10, ALUOp=00
T3	M[A+sigEx(offset)] ← B	IorD=1, MemWrite=1, T0

R -

Machine state	Operation	Control signals
T2	A Op B	ALUSrcA=1, $ALUSrcB_{1:0}$ = 00, ALUOp=00
T3	RF[dstn] ← A Op B	RegDst=1, MemtoReg=0, RegWrite=1, T0

B -

Machine state	Operation	Control signals
T1	(PC+4) + SigExtn(offset)	ALUSrcA=0, $ALUSrcB_{1:0}$ = 11, ALUOp=00
T2	A-B	ALUSrcA=1, $ALUSrcB_{1:0}$ = 00, ALUOp=01, Branch=1, T0

Decode stage

ADDI -

Machine state	Operation	Control signals
T2	A OP SigExtn(offset)	ALUSrcA=1, $ALUSrcB_{1:0}$ = 10, ALUOp=00
T3	RF[Destn]← A OP SigExtn(offset)	RegDst=0, MemtoReg=0, T0

Jump –

Machine state	Operation	Control signals
T2	A OP SigExtn(offset)	Jump=1, T0

$$T_c = t_{pcq\_PC} + t_{mux} + \max\{t_{ALU} + t_{mux}, t_{mem}\} + t_{registerRead}$$

T_c is going to decide the Clock time period

Multi-cycle processor is less expensive

It has 5-nonarchitectural elements

Pipeline :

- Cost/performance ratio

$$\frac{C}{P} = \frac{G + k * L}{\frac{1}{(\frac{T}{k} + S)}}$$

$$= LT + GS + LSk + \frac{GT}{k}$$

$$k_{opt} = \sqrt{\frac{GT}{LS}}$$

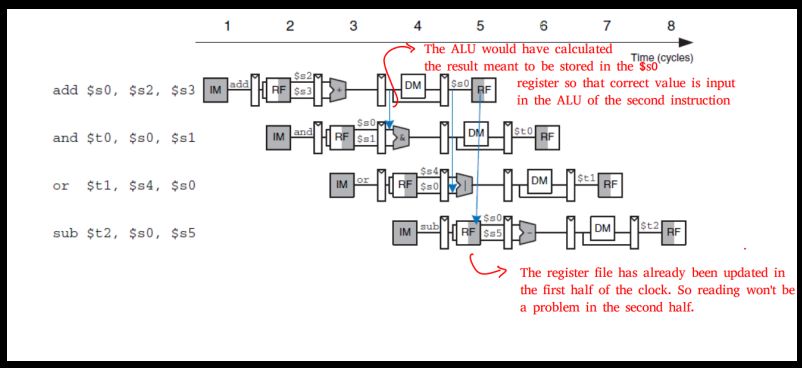
↳ Optimal no. of latches.

- Clock period (T): $T_2 - T_1 > T_M - T_m + T_L$

Data Hazard →

a) RAW without LW →

2 types of Forward — (i) MEM to execute stage
(ii) WB stage to execute stage.



Data Hazards: H/W-based Solution

25

- Hazard unit generates control signals for
  - Mux SrcA [ForwardAE]
  - Mux SrcB [ForwardBE]
- The control logic for ForwardAE
- if ((rsE!=0) AND (rsE == WriteRegM) and RegWriteM) then
  ForwardAE = 10 → ALUOut goes in
- else if ((rsE!=0) AND (rsE == WriteRegW) and RegWriteW) then
  ForwardAE = 01 → Output from Data Memory goes in.
- else
  ForwardAE = 00 → rs, rt ftw.

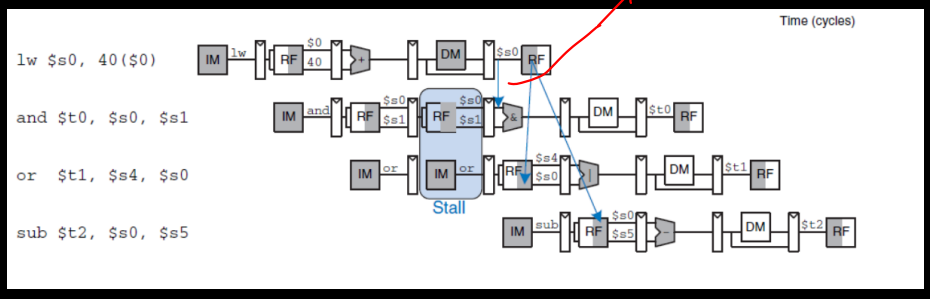
Write Reg is register to be written in register file (see in latch of MEM stage)

See in the latch storing info for MEM stage

See in latch of write stage

b) Raw with LW — Stalling is required

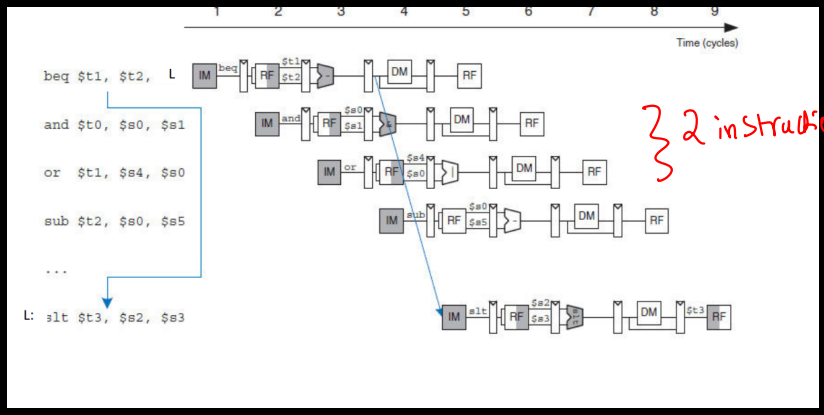
Appropriate forwarding also needs to be done



- lwstall = ((rsD == rtE) OR (rtD == rtE)) AND MemtoRegE
- StallF = lwstall Switch off the program counter
- StallD = lwstall Switch off the latch after instruction stage
- FlushE = lwstall Clear the values in the latch after decode stage

Control Hazard: Happens due to Branch instructions.

1)



} 2 instructions needed to be flushed in this case if Branch is taken.

2) Modifications —

- Stall: Waste of cycle
- Add a Comparator in ID stage: Branch instruction exits here itself.

} Both causes only 1 instrn to be flushed

We use this from now on.

{ Both ZERO, Br. Address are calc. in ID stage }

$$\text{Speedup} = \frac{CPI\ unpipelined}{1 + pipeline\ stall\ cycle\ from\ branchs} = \frac{\frac{Pipeline\ depth}{1 + Branch\ frequency \times Branch\ penalty}}{} = \frac{T_{unpipe}}{T_{pipe}}$$

Another Problem: —

ADD R1, R2, R3

BEQ R1, R2, offset

IF → ID → Ex → MEM → WB

IF → ID → - - -

↳ completes here.

But since the R1=R2+R3 value is not available until end of 'Ex', forwarding of this value can't be done at the start of 'Ex' state.

Also, we know BEQ would be making use of this value in the ID stage itself. ⇒

Solution: Stall the branch instruction. After that forward it.

- FowardAD = (rsD != 0) AND (rsD == WriteRegM) AND RegWriteM
- FowardBD = (rtD != 0) AND (rtD == WriteRegM) AND RegWriteM

What if one of the source operand is in Memory stage (lw-type instr.)

- Branchstall=

BranchD AND RegWriteE AND (WriteRegE == rsD OR WriteRegE == rtD)

OR

BranchD AND MemtoRegM AND (WriteRegM == rsD OR WriteRegM == rtD)

- StallF = StallD = FlushE = lwstall OR branchstall

→ Some kind of stall in both cases

- Cycle period, $T_c = max \begin{pmatrix} t_{pcq} + t_{mem} + t_{setup} \\ 2(t_{RFread} + t_{mux} + t_{eq} + t_{AND} + T_{mux} + t_{setup}) \\ t_{pcq} + t_{mux} + t_{mux} + t_{ALU} + t_{setup} \\ t_{pcq} + t_{memwrite} + t_{setup} \\ 2(t_{pcq} + t_{mux} + t_{RFWrite}) \end{pmatrix}$

Predictions: Whenever a B-type instruc^n actually branches, an instruction has to be flushed. To avoid this, the compiler can take steps based on the prediction it makes.

If prediction is — [Branch untaken] : Would execute normally. (Suitable for forward type branch)

If pred^n is — [Branch Taken] : —

- Rearrange the code such that the next instruction is the one that is not dependent on the Branch block. Like:

```
if ( . . )
{
}
else
{
}
```

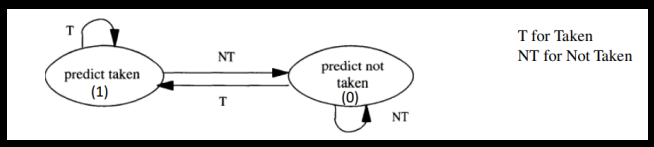
This instruction → x y z would surely take place

So, execute it after branch itself.

Delayed Branch Technique.

- There is a prediction bit in the Branch Target Buffer (in the cache) when this bit could be changed by the compiler.

This Buffer is accessed in the IF stage itself and the prediction is made.



Memory Hierarcy →

1. Spatial : Neighbors being accessed
   Temporal : Same element being accessed
2. → CPU provides address for the memory block required to be accessed along with the offset.

→ Each block would contain Multiple words.
   ↳ 4 bytes = 32 bits.
   No. of words = block size = b.
   No. of blocks = No. of cache lines present in the cache = S.
   So, cache capacity = — $C = b * S$

- A cache has the following parameters: b, block size given in numbers of words; S, number of blocks; N, number of ways, and A, number of address bits.
- In terms of the parameters described, what is the cache capacity, C?
- C=b*S
- In terms of the parameters described, what is the total number of bits required to store the tags?
- No of bits for Tag = A – $\log_2(b)$ – $\log_2$ (S/N)
- What are S and N for a direct mapped and fully associative cache of size C words and block size b?
- S=C/b, N=S for fully associative cache
- S=C/b, N=1 for direct mapped cache

— x —

Interaction Policies of Cache : —

CPU → Cache
↓
Main Mem

1. Read : —
   a) Look Aside (Parallel) → Both CPU and Cache see the Address Bus at the same time.
   → If cache hit, then CPU takes info from here and bus cycle is terminated
   → If miss, take info from Main mem and write it in cache for the next time.

- It is less complex and less expensive
- Provide better response to a cache miss
- Drawback is the processor cannot access cache while another bus master is accessing main memory

   b) Look Through (Series) —
   if (hit) : {take info from cache ; don't send any request to Main mem}
   else : { send req to main mem ; store info to cache also after reading }

Write Policies:

Write hit policy	Write miss policy
Write Through	Write Allocate
Write Through	**No Write Allocate**
Write Back	**Write Allocate**
Write Back	No Write Allocate

- ***Write Through*** - the information is written to both the block in the cache and to the block in the lower-level memory.

Advantage:
- read miss never results in writes to main memory
- easy to implement
- main memory always has the most current copy of the data (consistent)

Disadvantage:
- write is slower
- every write needs a main memory access
- as a result uses more memory bandwidth

Related

- ***Write back*** - the information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced. To reduce the frequency of writing back blocks on replacement, a ***dirty bit*** is commonly used. This status bit indicates whether the block is dirty (modified while in the cache) or clean (not modified). If it is clean the block is not written on a miss.

Advantage:
- writes occur at the speed of the cache memory
- multiple writes within a block require only one write to main memory
- as a result uses less memory bandwidth

Disadvantage:
- harder to implement
- main memory is not always consistent with cache
- reads that result in replacement may cause writes of dirty blocks to main memory

if cache is full and some line is being replaced, then dirty blocks must be written.

Write Through with Write Allocate:
- on hits it writes to cache and main memory
- on misses it updates the block in main memory and brings the block to the cache
- Bringing the block to cache on a miss does not make a lot of sense in this combination because the next hit to this block will generate a write to main memory anyway (according to Write Through policy)

Write Through with No Write Allocate:
- on hits it writes to cache and main memory;
- on misses it updates the block in main memory not bringing that block to the cache;
- Subsequent writes to the block will update main memory because Write Through policy is employed. So, some time is saved not bringing the block in the cache on a miss because it appears useless anyway.

Write Back with Write Allocate:
- on hits it writes to cache setting dirty bit for the block, main memory is not updated;
- on misses it updates the block in main memory and brings the block to the cache;
- Subsequent writes to the same block, if the block originally caused a miss, will hit in the cache next time, setting dirty bit for the block. That will eliminate extra memory accesses and result in very efficient execution compared with Write Through with Write Allocate combination.

Write Back with No Write Allocate:
- on hits it writes to cache setting dirty bit for the block, main memory is not updated;
- on misses it updates the block in main memory not bringing that block to the cache;
- Subsequent writes to the same block, if the block originally caused a miss, will generate misses all the way and result in very inefficient execution.