Pipeline II
Hazards & Branch Detect

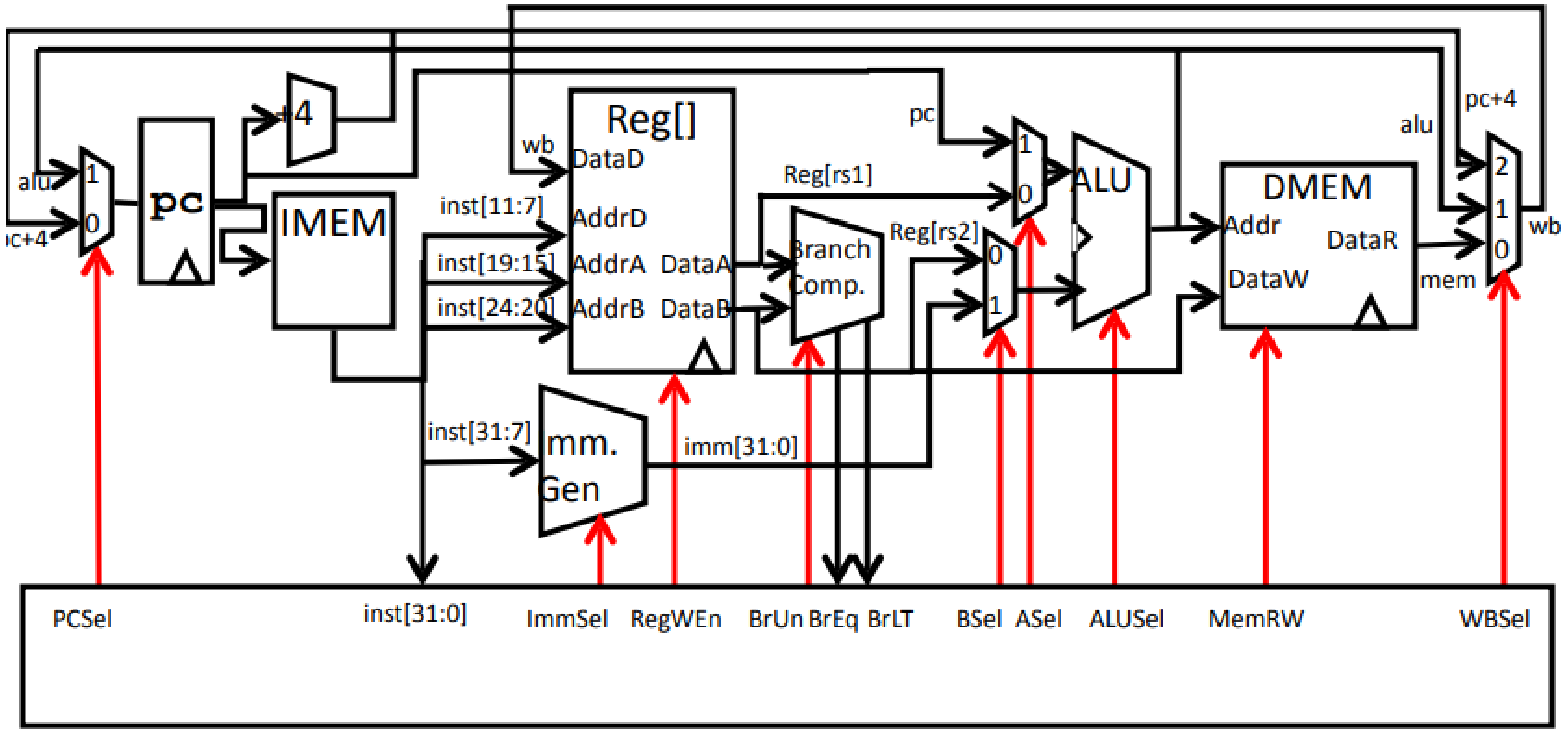
Chixiao Chen

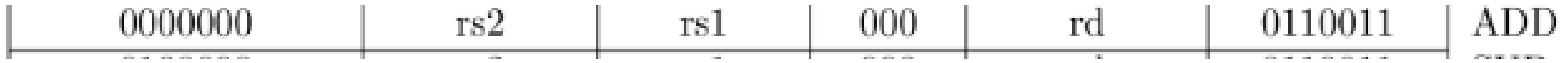
Coutesy by

Overview

- Review on the Single Cycle Processor
- Data hazards
  - Why ? Register file timing
  - R-type + R-type
  - Load + R-type
- Control hazard
- AI Perspective

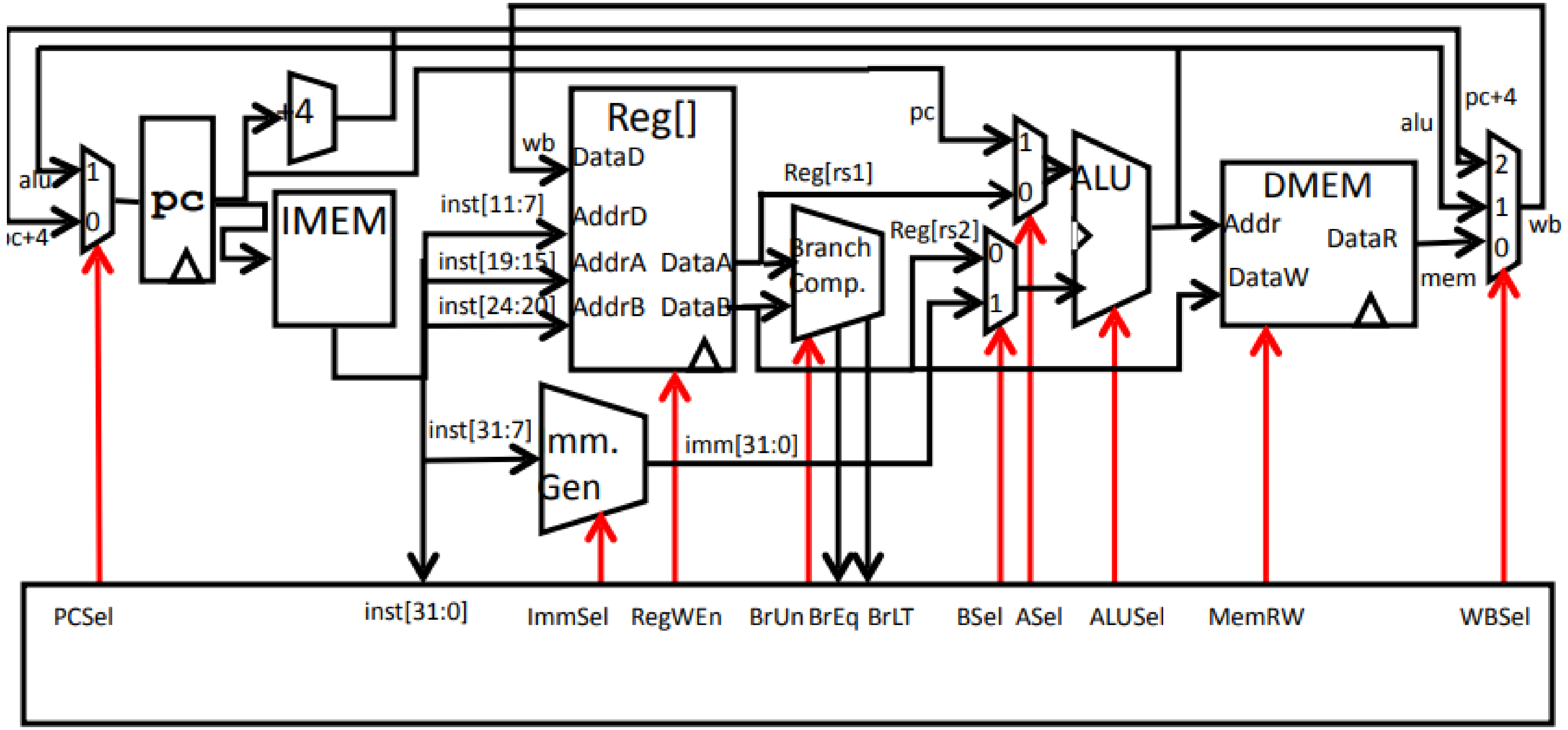
Overall Single Cycle RV32I Datapath





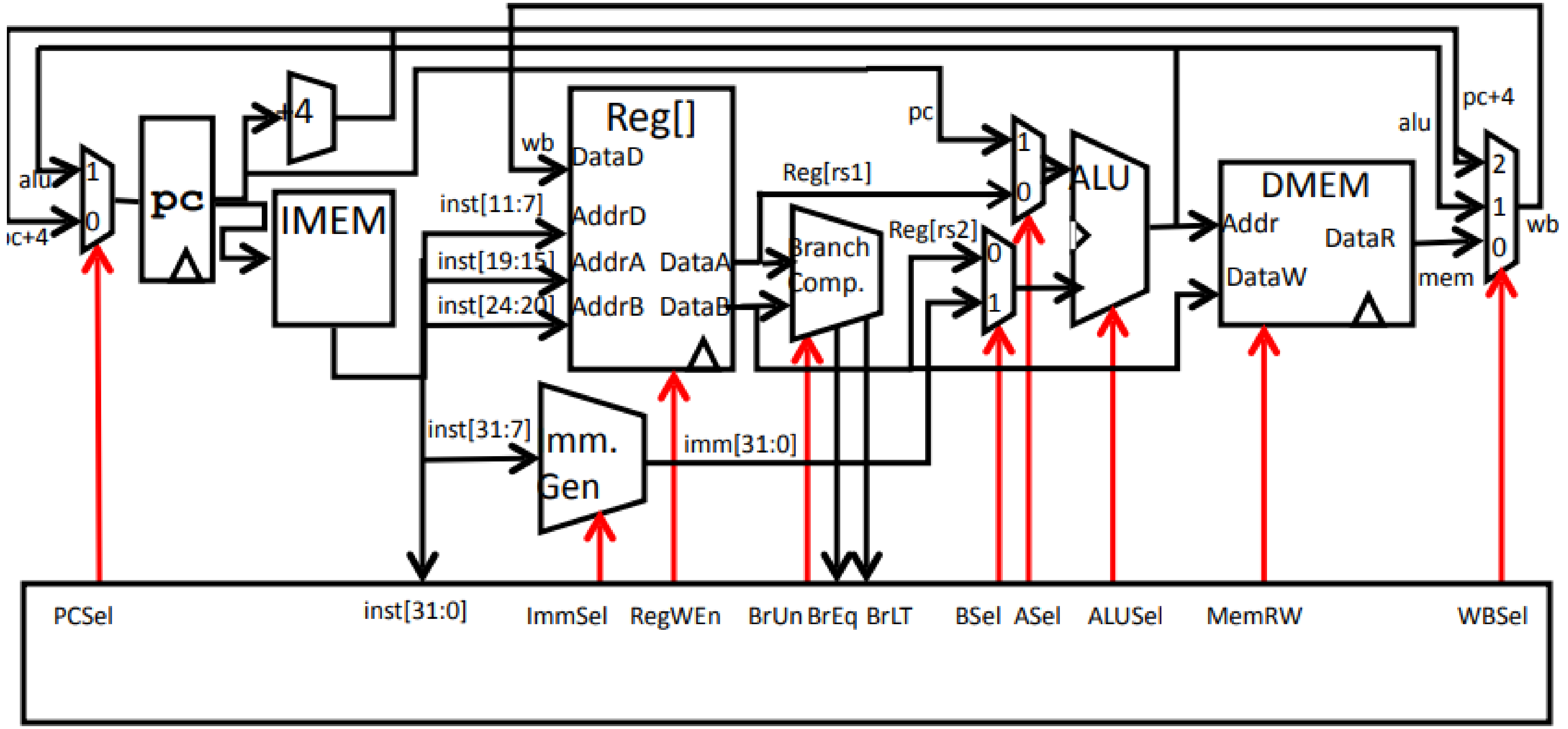
0000000	rs2	rs1	000	rd	0110011	ADD

Overall Single Cycle RV32I Datapath



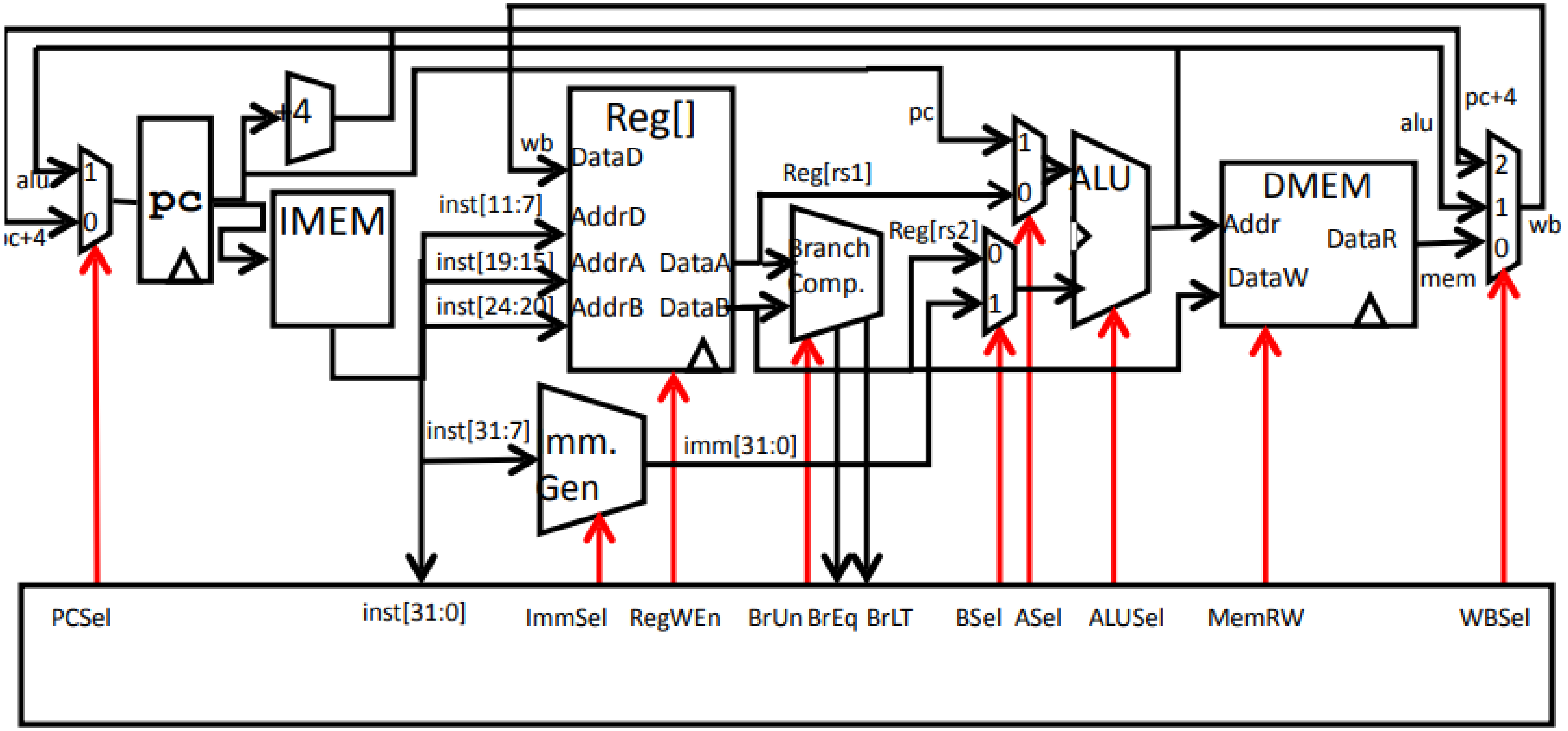
imm[11:0]	rs1	010	rd	0000011	LW

Overall Single Cycle RV32I Datapath



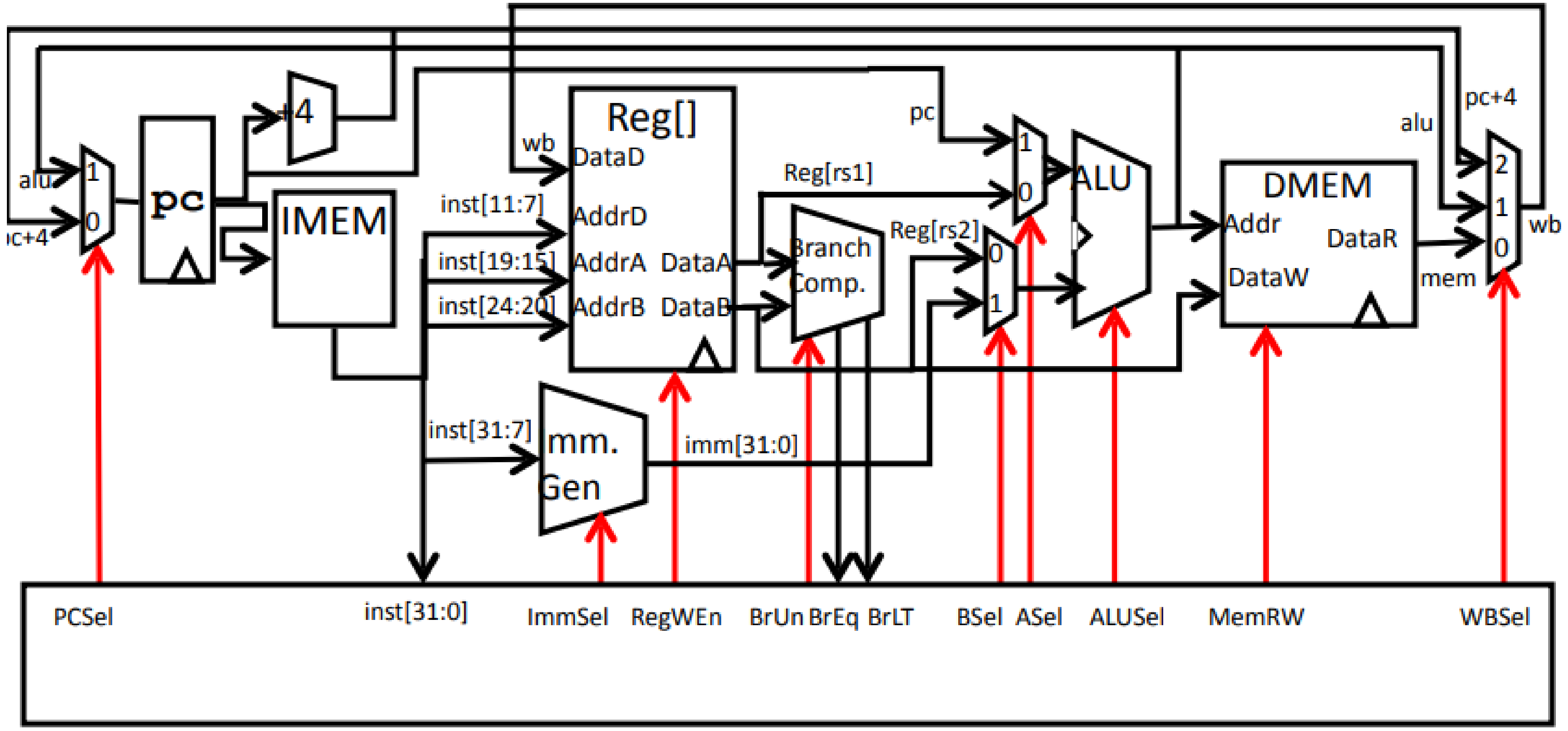
31 … 25	24 … 20	19 … 15	14 … 12	11 … 7	6 … 0
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7	5	5	3	5	7
offset[11:5]	src	base	width	offset[4:0]	STORE

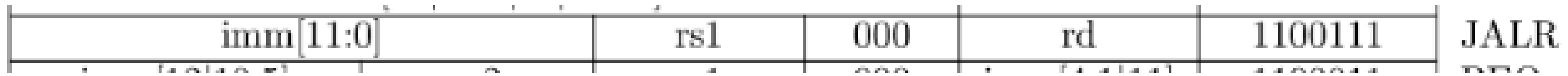
Overall Single Cycle RV32I Datapath



31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
1	6	5	5	3	4	1	7

Overall Single Cycle RV32I Datapath





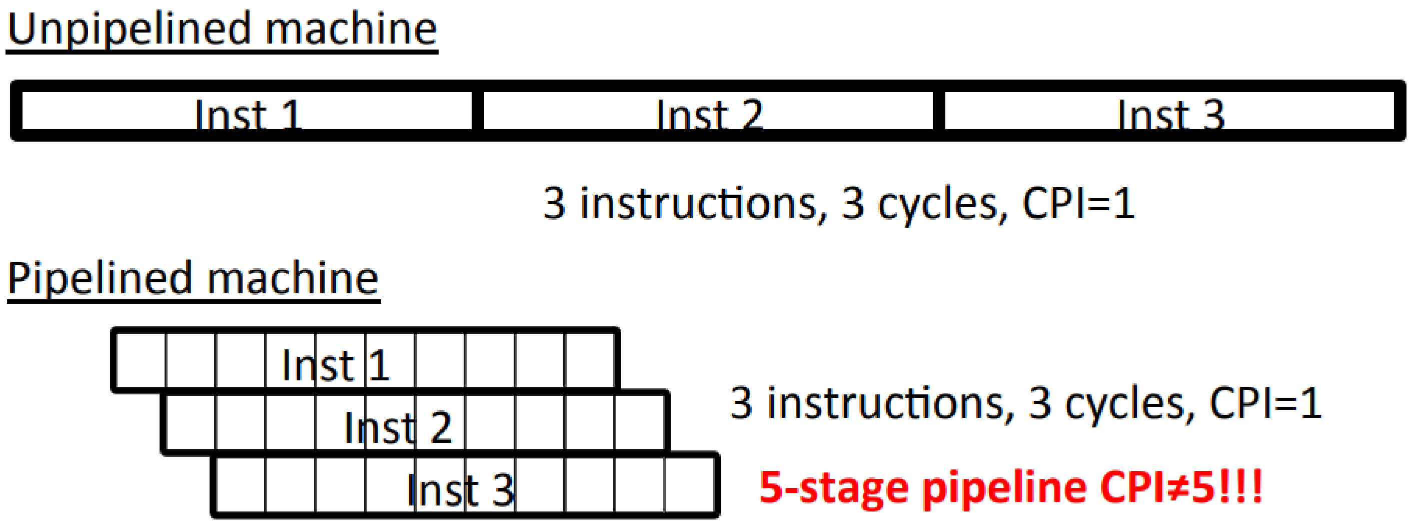
imm[11:0]	rs1	000	rd	1100111	JALR

Throughput

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Cycles}}{\text{Instruction}} * \frac{\text{Time}}{\text{Cycle}}$$

- Instructions per program depends on source code, compiler technology, and ISA .
- Cycles per instructions (CPI) depends on ISA and μarchitecture
- Time per cycle depends upon the μarchitecture and base technology

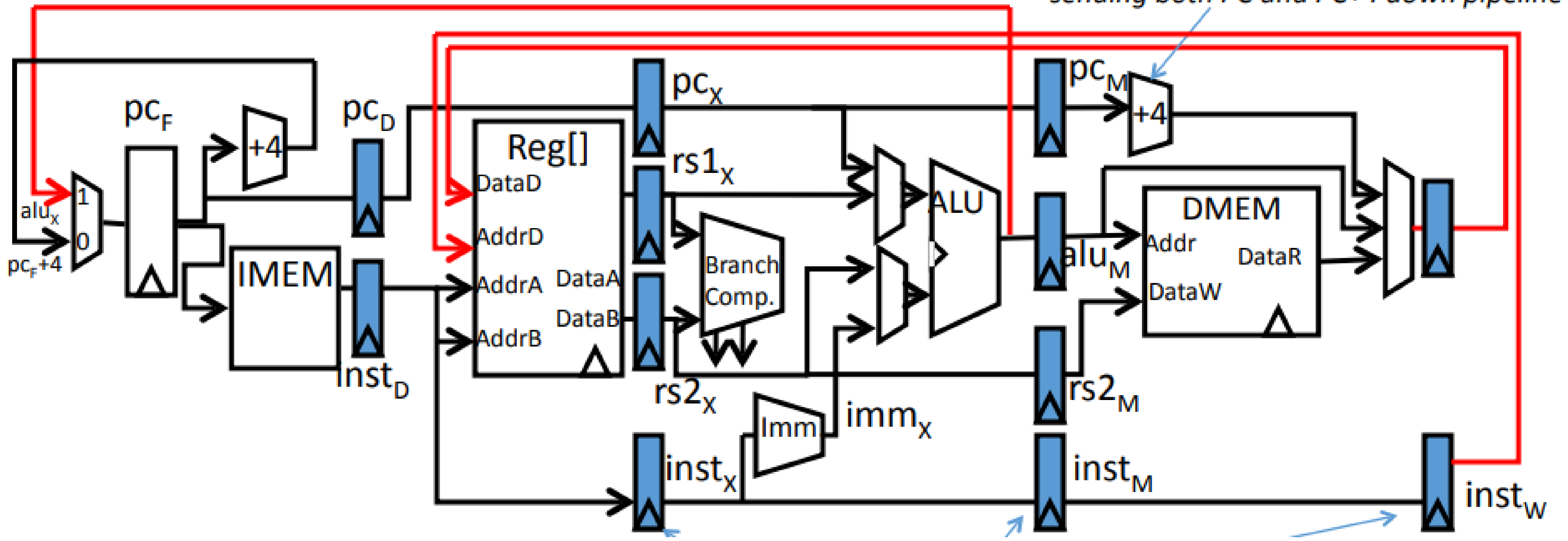
- Why pipeline is good?



5-stage pipeline CPI≠5!!!

Pipelined RV32-I

Recalculate PC+4 in M stage to avoid sending both PC and PC+4 down pipeline



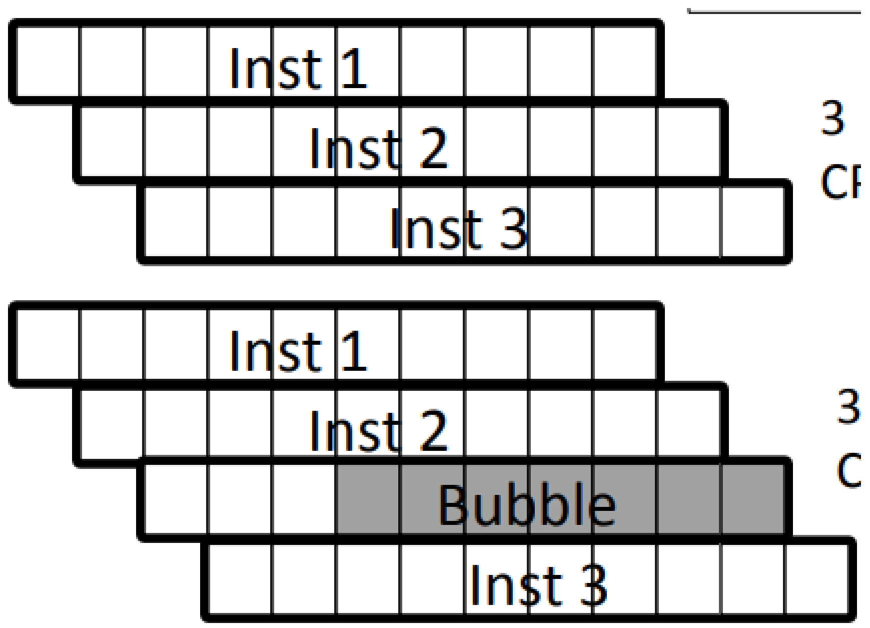
Hazard

Structural / Data / Control

Hazards

Instructions interact with each other in pipeline

- An instruction in the pipeline may need a resource being used by another instruction in the pipeline → *structural hazard*
- An instruction may depend on something produced by an earlier instruction
  - Dependence may be for a data value
    → *data hazard*
  - Dependence may be for the next instruction's address
    → *control hazard (branches, exceptions)*

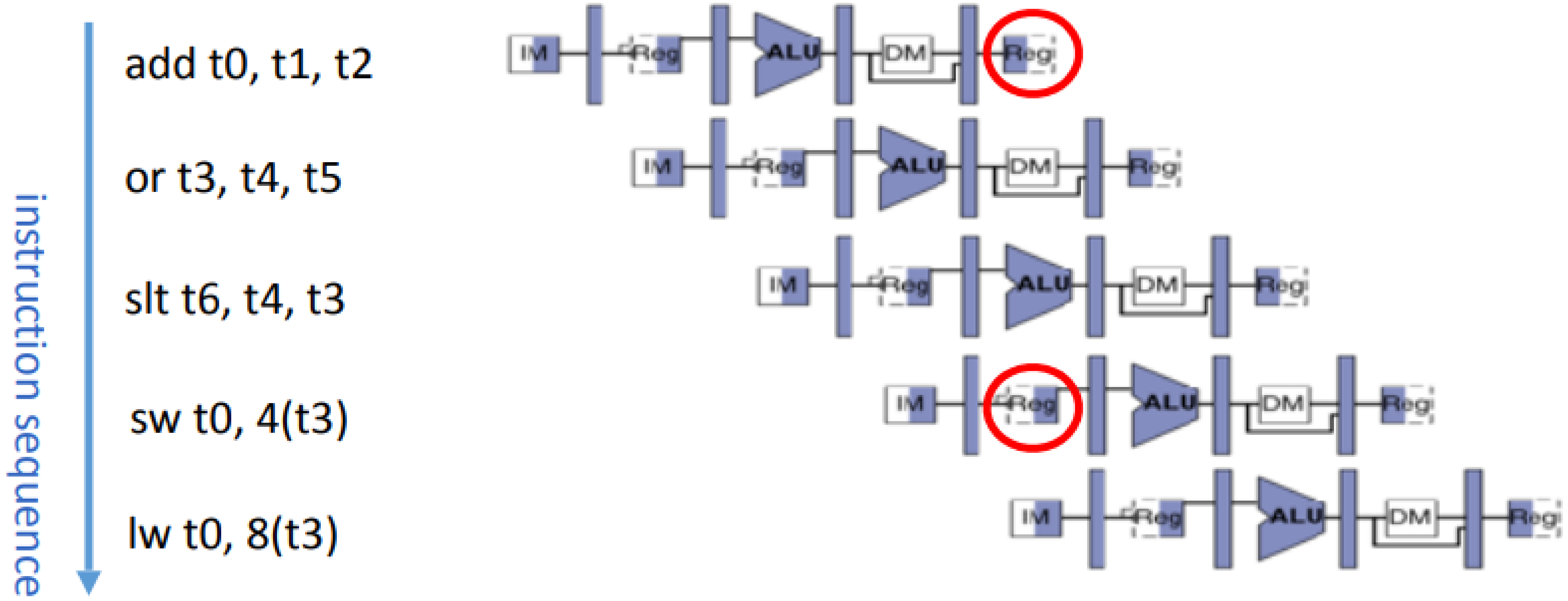


Structural Hazards

- Structural hazard occurs when two instructions need same hardware resource at same time
- A structural hazard can always be avoided by adding more hardware to design
  - E.g., if two instructions both need a port to memory at same time, could avoid hazard by adding second port to memory
- Does Classic RISC 5-stage integer pipeline have structural hazards by design ?

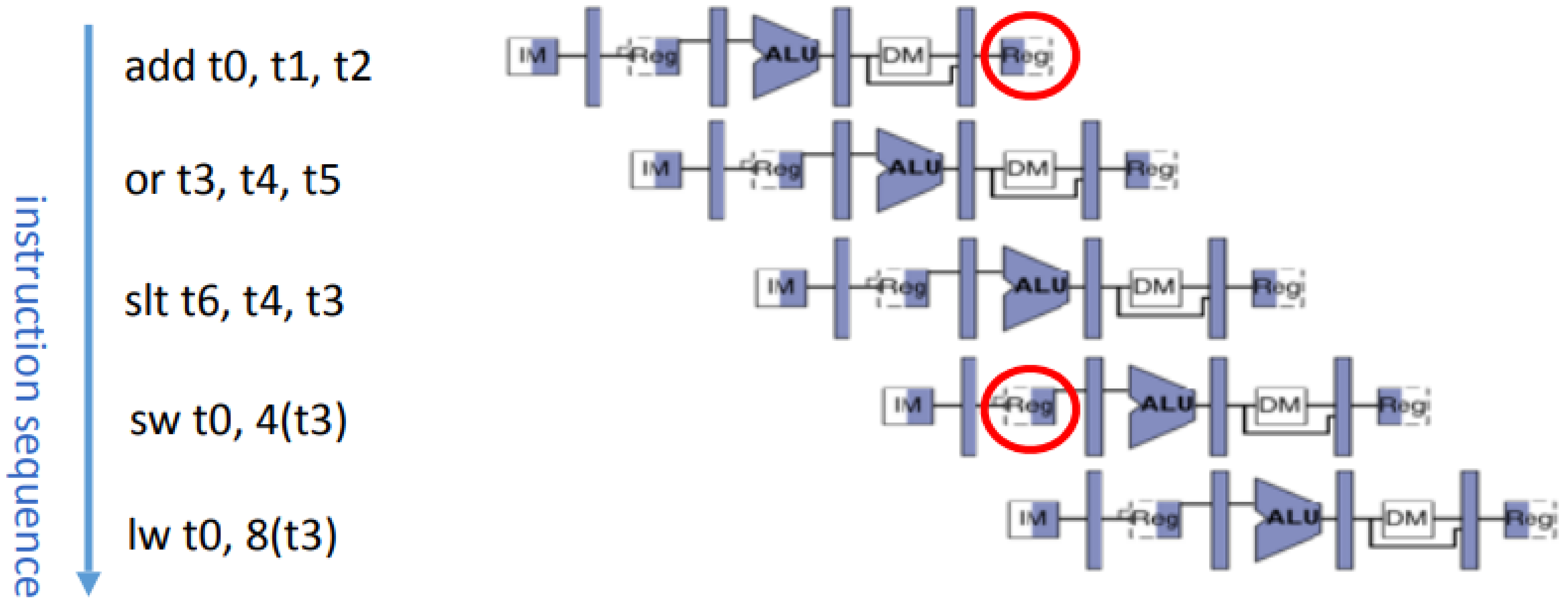
Register Access

- Two instructions using Reg file at same time
- If the data is same, also data hazards



Register Access

- Exploit high speed of register file
  - WB updates value @ posedge / ID reads new value @ negedge



Data Hazards

- Consider executing a sequence of register-register instructions of type:

  rk ← ri op rj

- In 5-stage pipeline, which one shall we care?

Data-dependence

$$\begin{tikzcd} r_3 \arrow[rr, shift right=3] & \leftarrow & r_1 \text{ op } r_2 \\ r_5 & \leftarrow & r_3 \text{ op } r_4 \end{tikzcd}$$

Read-after-Write (RAW) hazard

Anti-dependence

$$\begin{tikzcd} r_3 & \leftarrow & r_1 \text{ op } r_2 \arrow[dll] \\ r_1 & \leftarrow & r_4 \text{ op } r_5 \end{tikzcd}$$

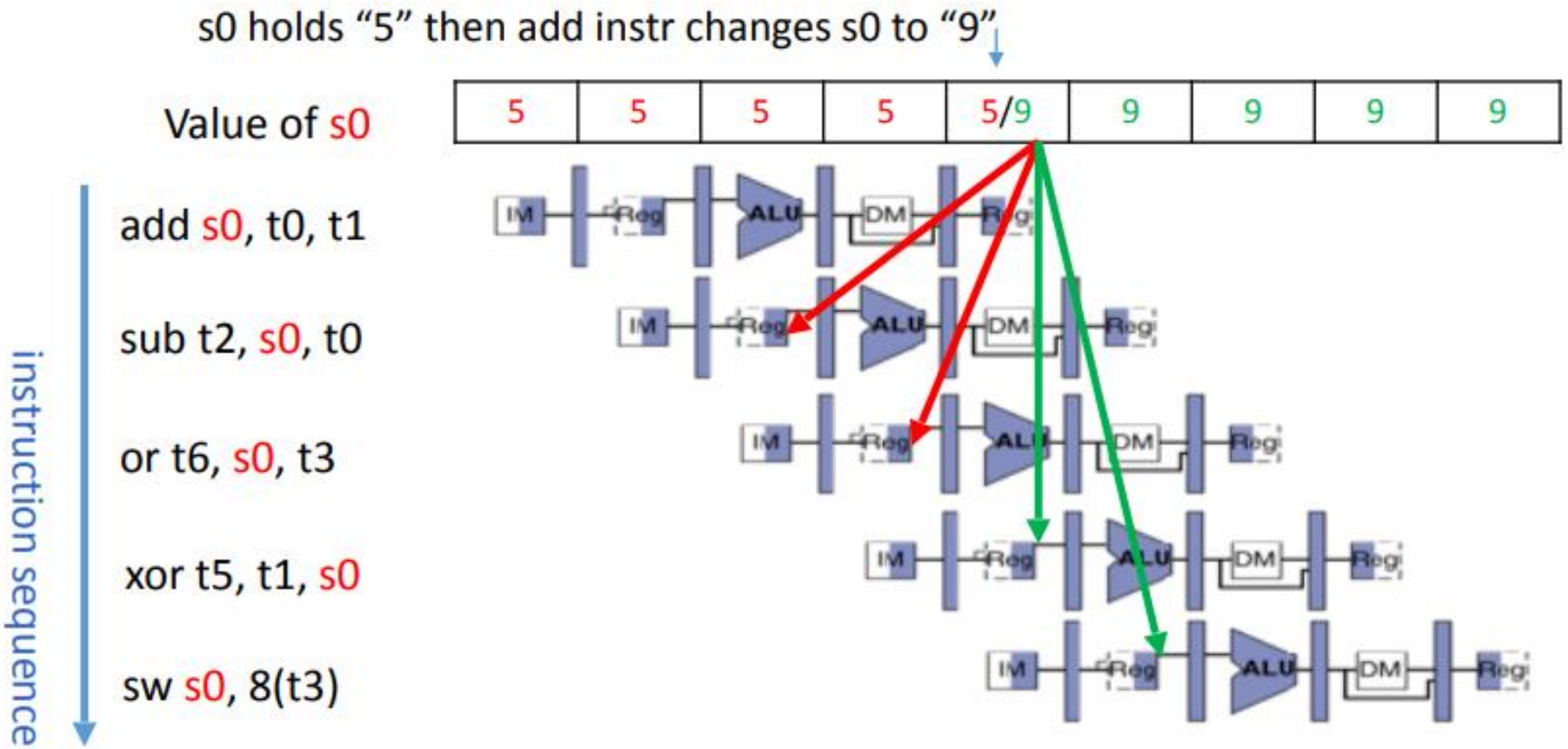
Write-after-Read (WAR) hazard

Output-dependence

$$\begin{tikzcd} r_3 \arrow[d, bend right] & \leftarrow & r_1 \text{ op } r_2 \\ r_3 & \leftarrow & r_6 \text{ op } r_7 \end{tikzcd}$$

Write-after-Write (WAW) hazard

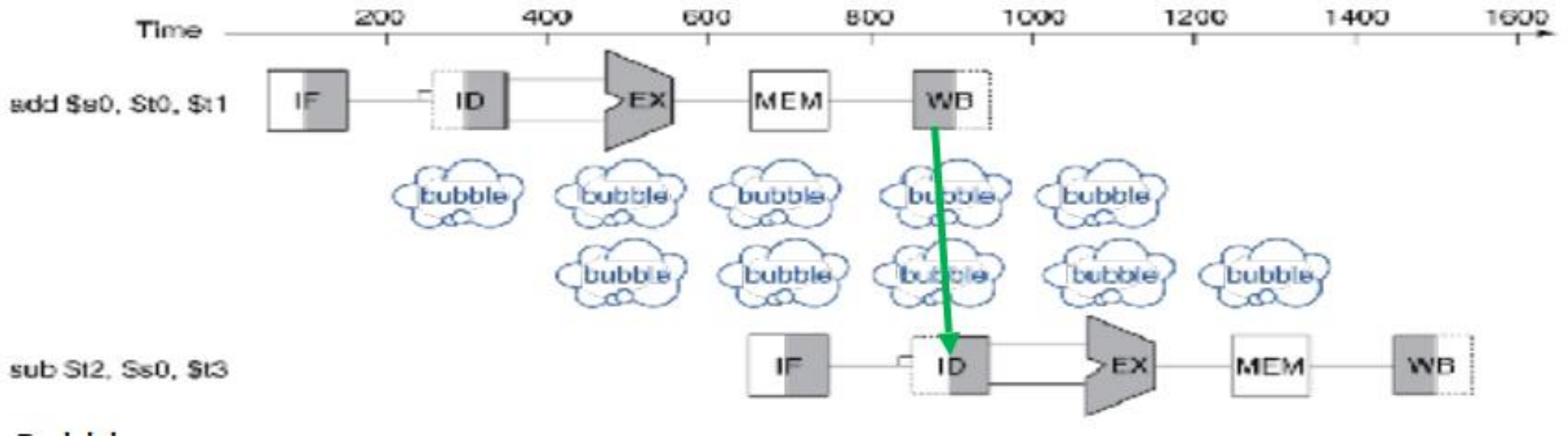
RAW example



s0 holds “5” then add instr changes s0 to “9”
Value of s0
5
5
5
5
5/9
9
9
9
9
instruction sequence
add s0, t0, t1
sub t2, s0, t0
or t6, s0, t3
xor t5, t1, s0
sw s0, 8(t3)
IM
Reg
ALU
DM
Reg

Stall: Simple but not good

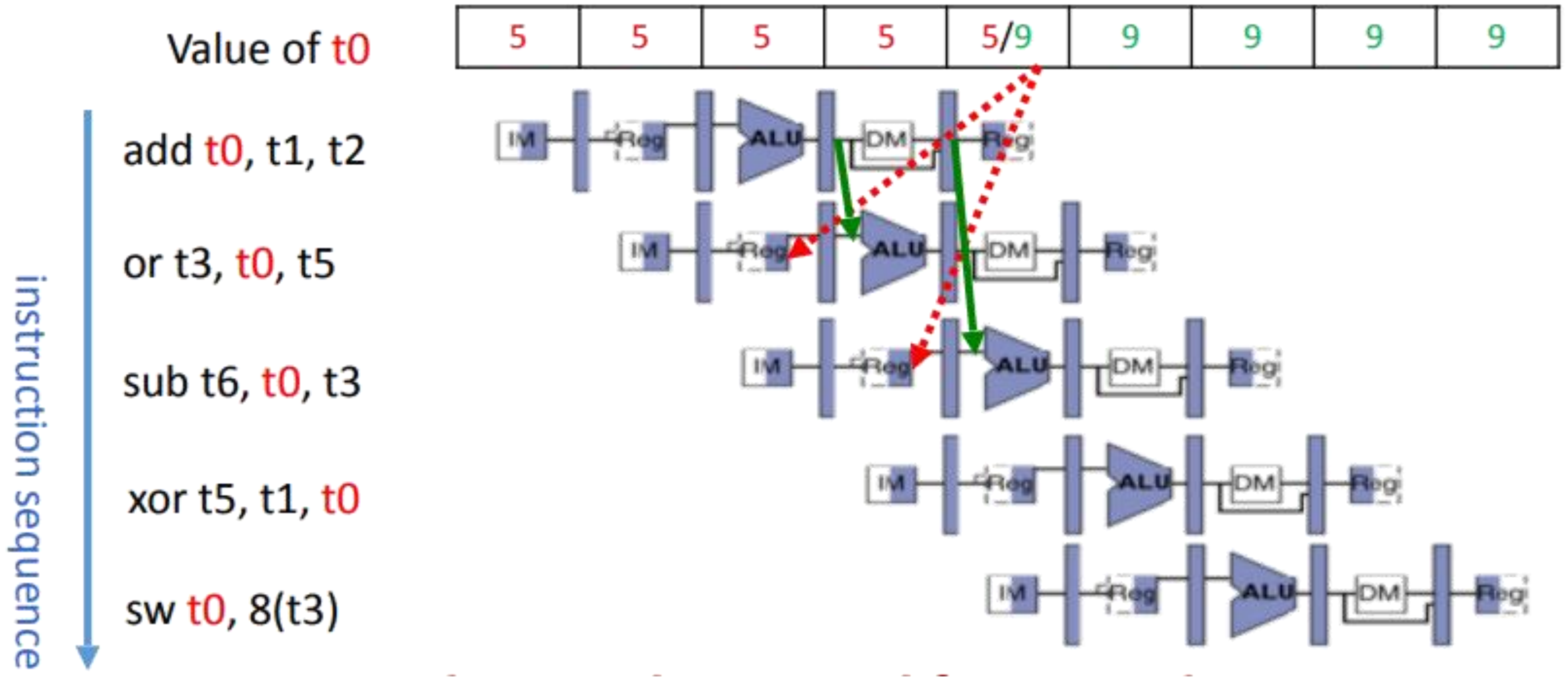
- Problem: Instruction depends on result from previous instruction
  - add s0, t0, t1
    sub t2, s0, t3



- Stalls reduce performance
- Compiler could try to arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure

Do we really need to wait ?

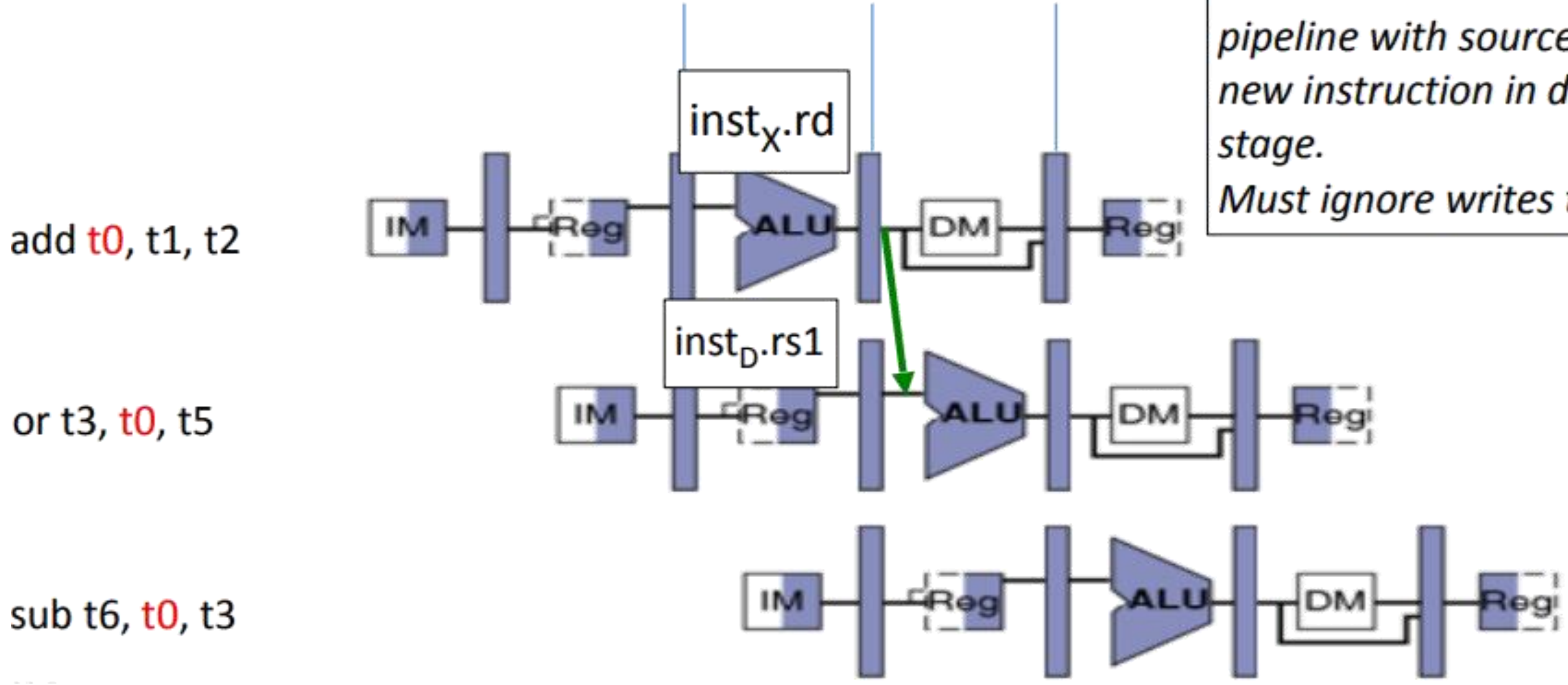
- If R-type, the value is calculated after ALU



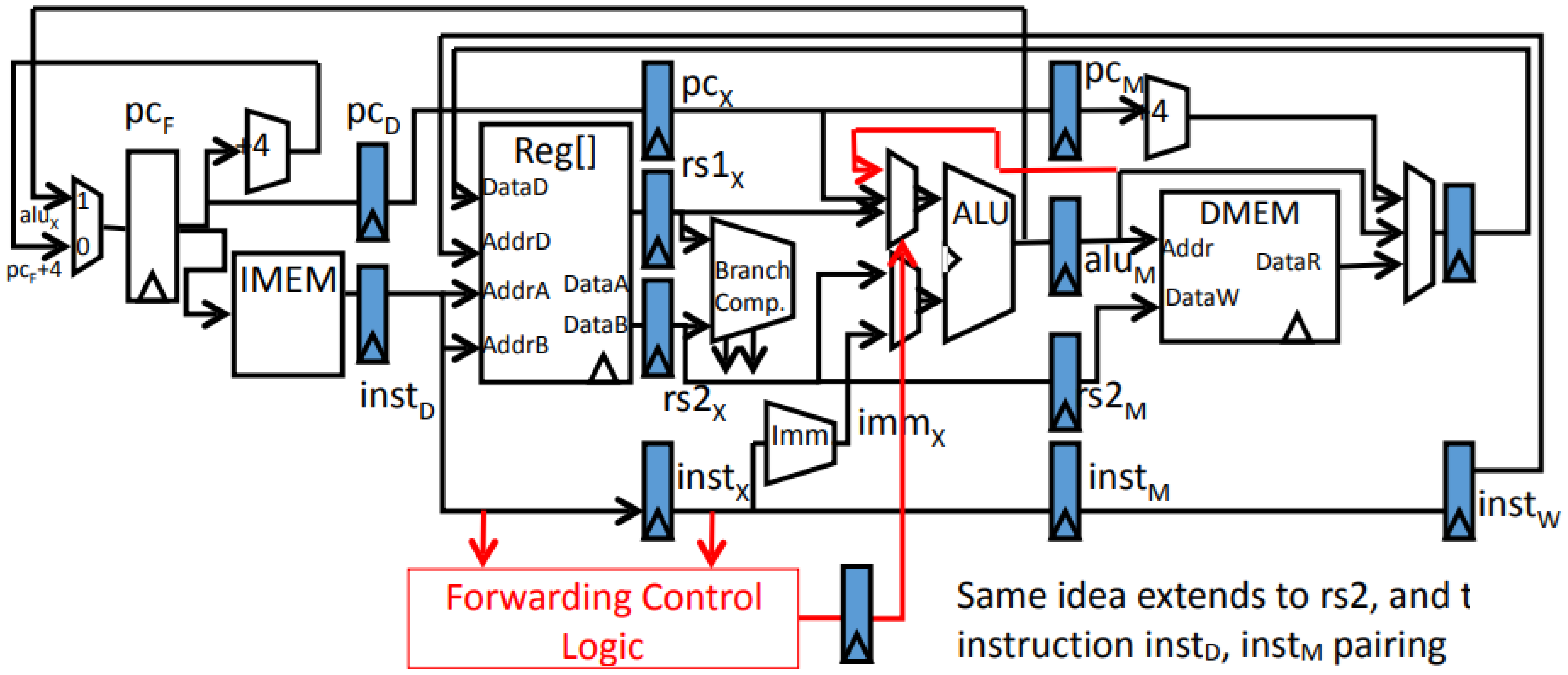
Forwarding / Bypassing

- Use result when it is computed
  - Don't wait for it to be stored in a register

Compare destination of older instructions in pipeline with sources of new instruction in decode stage.
Must ignore writes to x0!

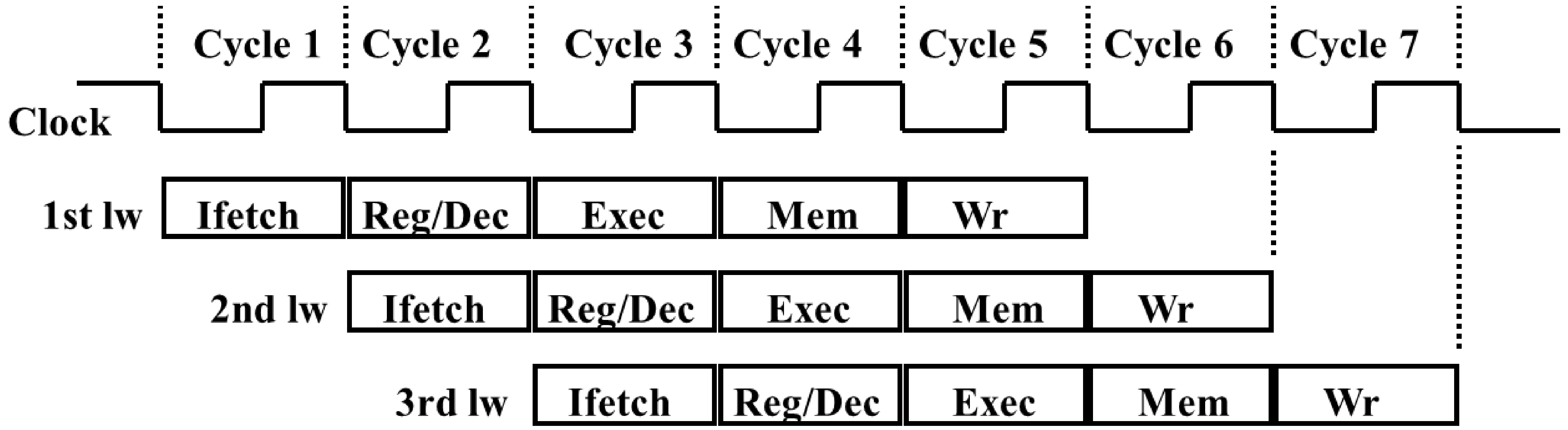


Hardware Modification



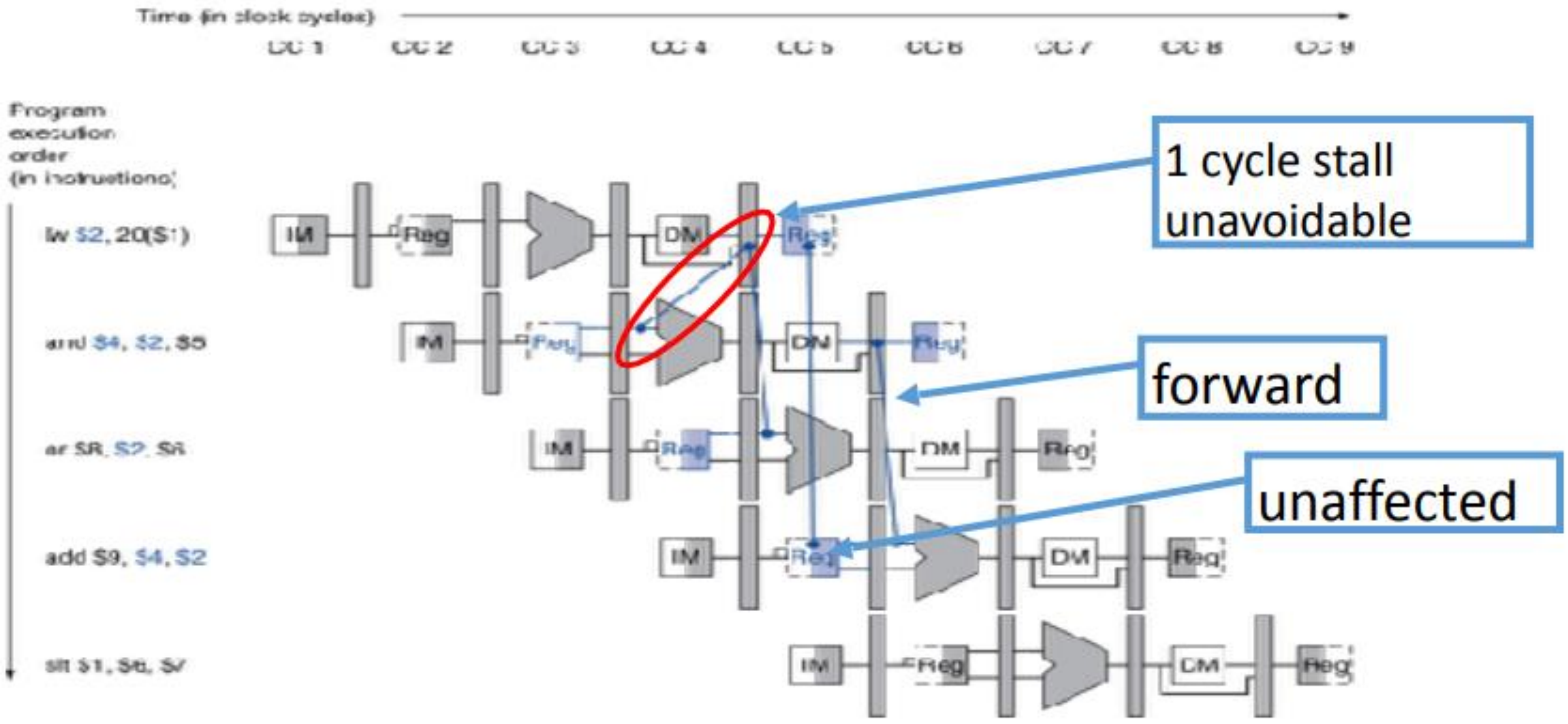
Same idea extends to rs2, and t
instruction inst$_D$, inst$_M$ pairing

Pipelining load



- 5 functional units in the pipeline datapath are:
  - ALU for the compute the address
  - Data Memory for get data from memery
  - Register File's Write port (busW) for the WB stage

Load data hazard



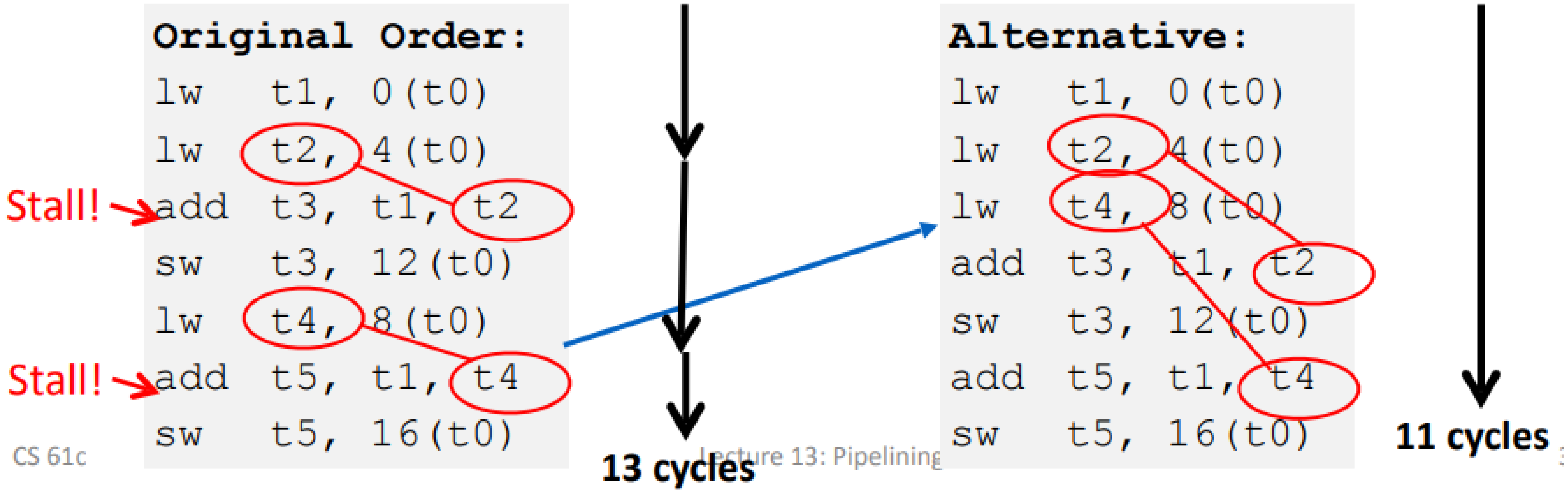
Stall pipeline

- Slot after a load is called a load delay slot
  - If that instruction uses the result of the load, then the hardware will stall for one cycle
  - Equivalent to inserting an explicit nop in the slot
  - Performance loss!
- Idea:
  - Put unrelated instruction into load delay slot
  - No performance loss!

Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction!

- RISC-V code for `D=A+B; E=A+C;`

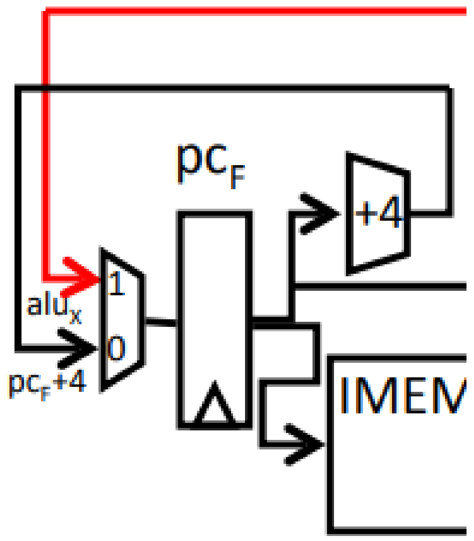


```
Original Order:
lw   t1, 0(t0)
lw   t2, 4(t0)
add  t3, t1, t2
sw   t3, 12(t0)
lw   t4, 8(t0)
add  t5, t1, t4
sw   t5, 16(t0)
```

```
Alternative:
lw   t1, 0(t0)
lw   t2, 4(t0)
lw   t4, 8(t0)
add  t3, t1, t2
sw   t3, 12(t0)
add  t5, t1, t4
sw   t5, 16(t0)
```

Control hazard

- When do we need to calculate next PC?
- For Unconditional Jumps
  - Opcode, PC, and offset
- For Jump Register
  - Opcode, Register value, and offset
- For Conditional Branches
  - Opcode, Register (for condition), PC and offset



Control Hazard Example -- Jump

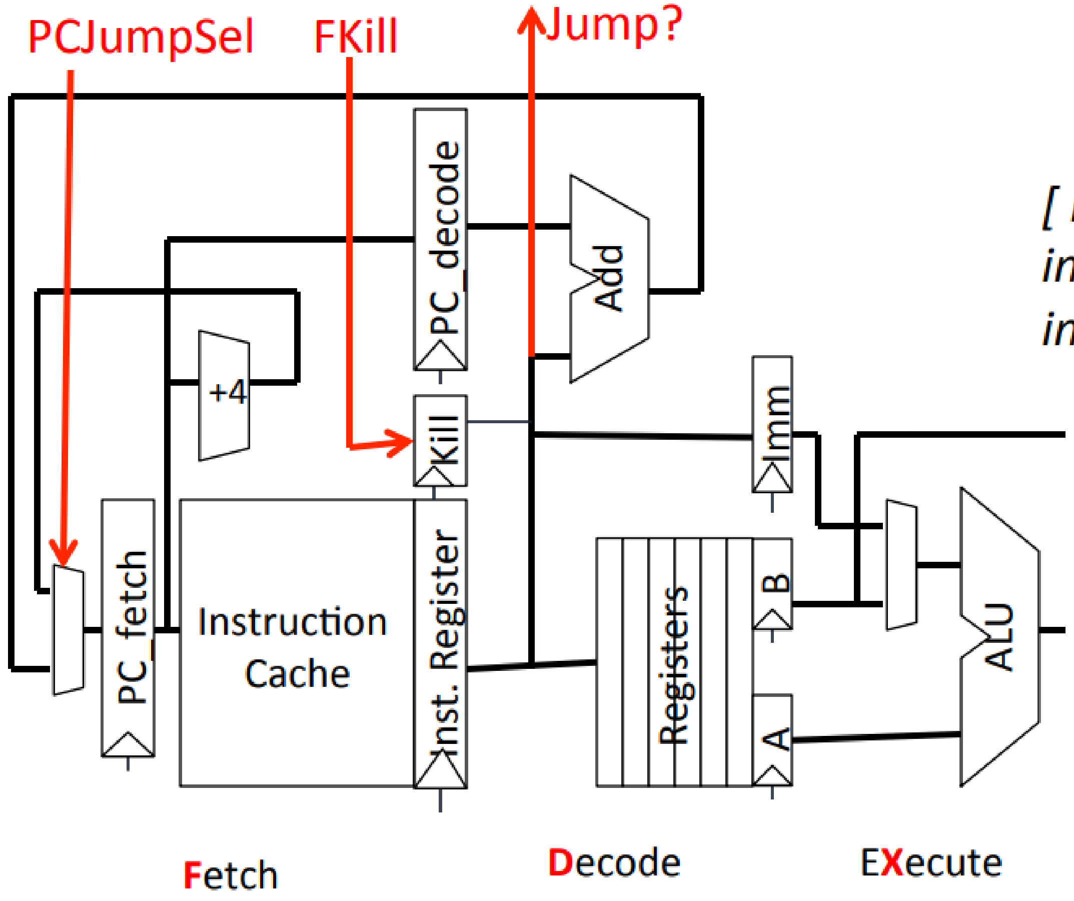
JAL Target

Target： add x1,x2,x3

Hardware modification

Insert kill logic in the control state registers.

Other method?



[Kill bit turns instruction into a bubble]

Control Hazard Example -- Branch

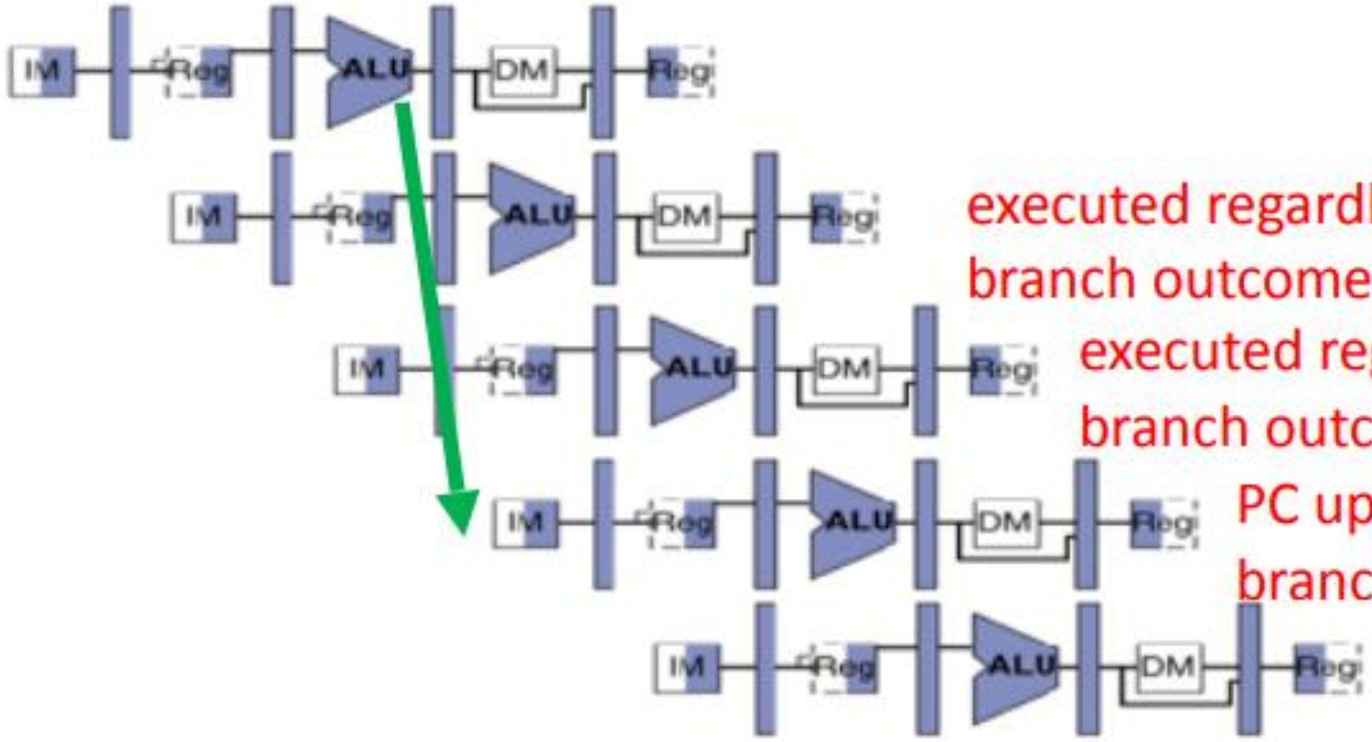
beq t0, t1, label

sub t2, s0, t5

or t6, s0, t3

xor t5, t1, s0

sw s0, 8(t3)



executed regardless of branch outcome!

executed regardless of branch outcome!!!

PC updated reflecting branch outcome

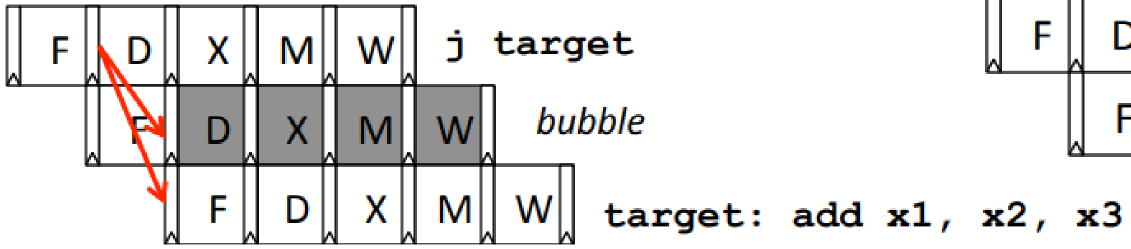
Strategy

- If branch not taken, then instructions fetched sequentially after branch are correct
- If branch or jump taken, then need to flush incorrect instructions from pipeline by converting to NOPs

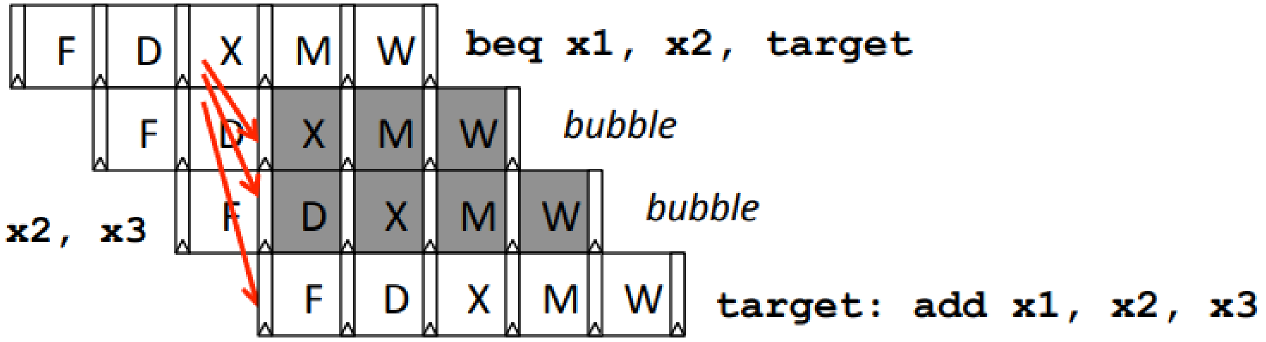
- How many branches need to be killed in the 5 stage RV32I architecture?

Question: jalr use which style

- JAL

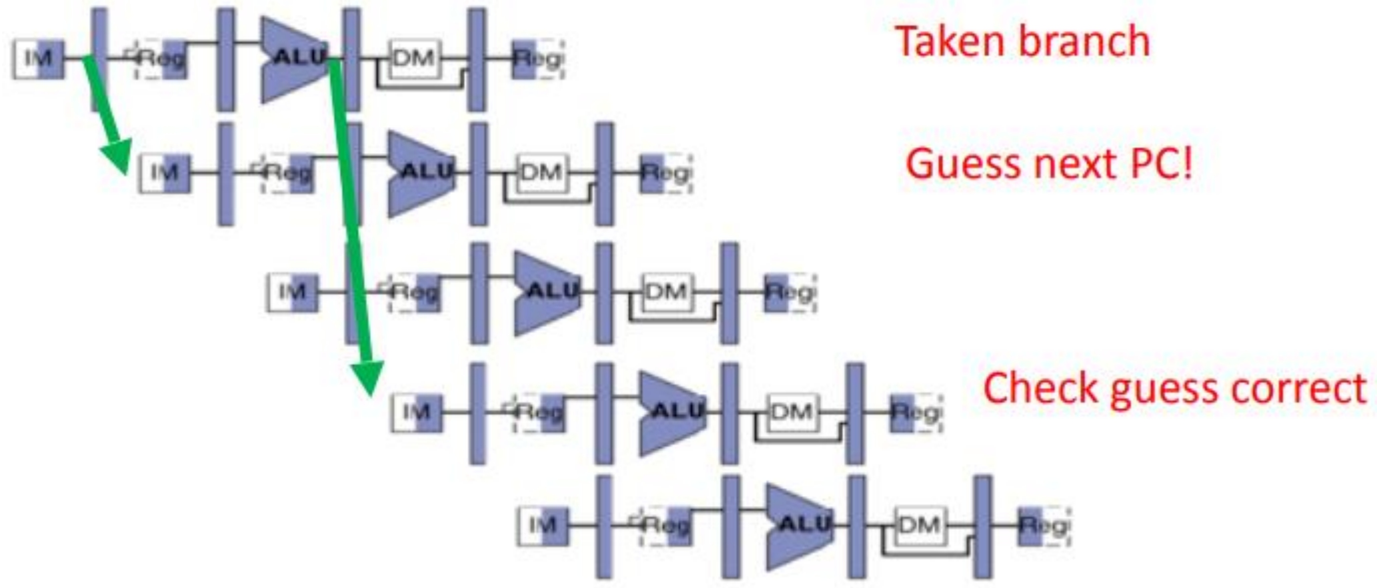


- Branch



Reducing Branch Penalties

- Every taken branch in simple pipeline costs 2 dead cycles
- To improve performance, use "branch prediction" to guess which way branch will go earlier in pipeline
- Only flush pipeline if branch prediction was incorrect



Branch Prediction

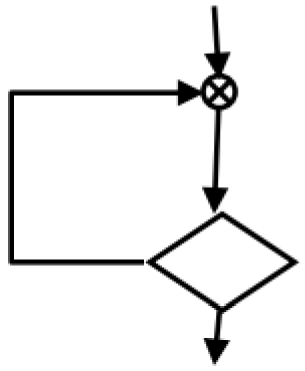
- Motivation:
  Branch penalties limit performance of deeply pipelined processors
  Modern branch predictors have high accuracy (>95%) and can reduce branch penalties significantly

- Required hardware support:
  - Prediction structures
  - Mispredict recovery mechanism

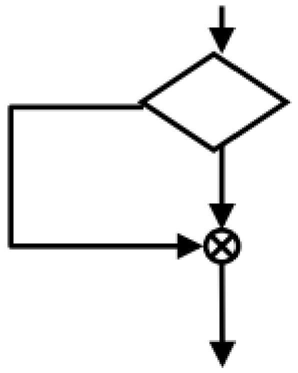
Static Branch Prediction

Overall probability a branch is taken is ~60-70% but:

backward
90%



forward
50%



ISA can attach preferred direction semantics to branches, e.g., Motorola MC88110

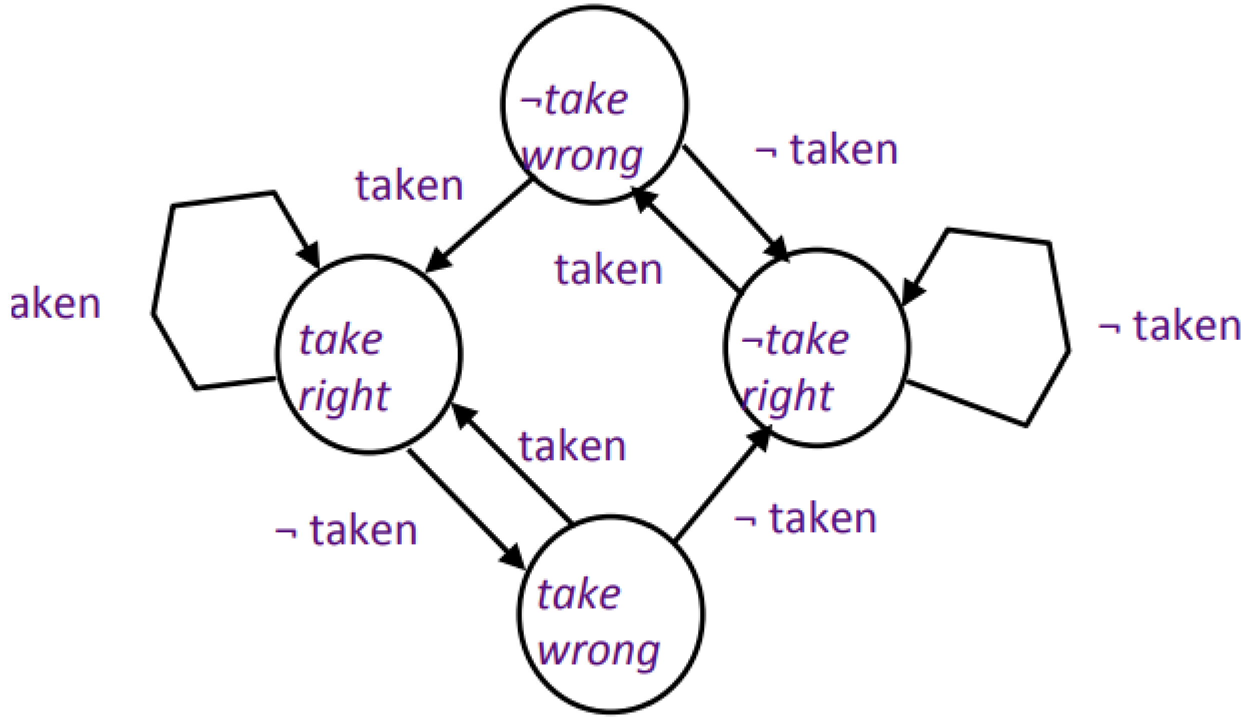
bne0 *(preferred taken)*   beq0 *(not taken)*

Dynamic Branch Prediction

- Learning based on past behavior
- Temporal correlation
  – The way a branch resolves may be a good predictor of the way it will resolve at the next execution
- Spatial correlation
  – Several branches may resolve in a highly correlated manner (a preferred path of execution)

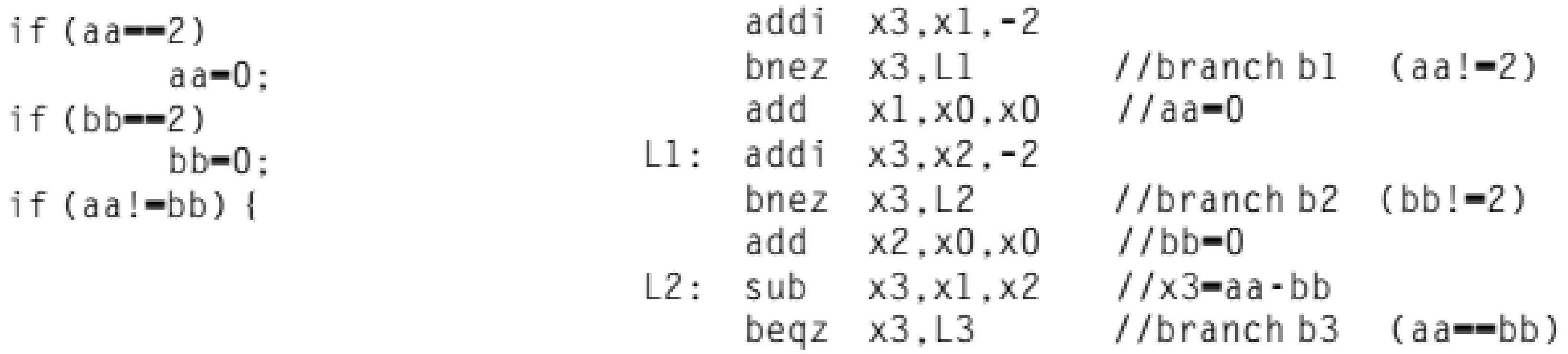
Temporal Correlation

- One bit history predictor
  - For each branch, remember last way branch went
- Two bit predictor
  - A finite state machine
  - Change the prediction after 2 mistackes



Spatial Correlation

- Other branch style

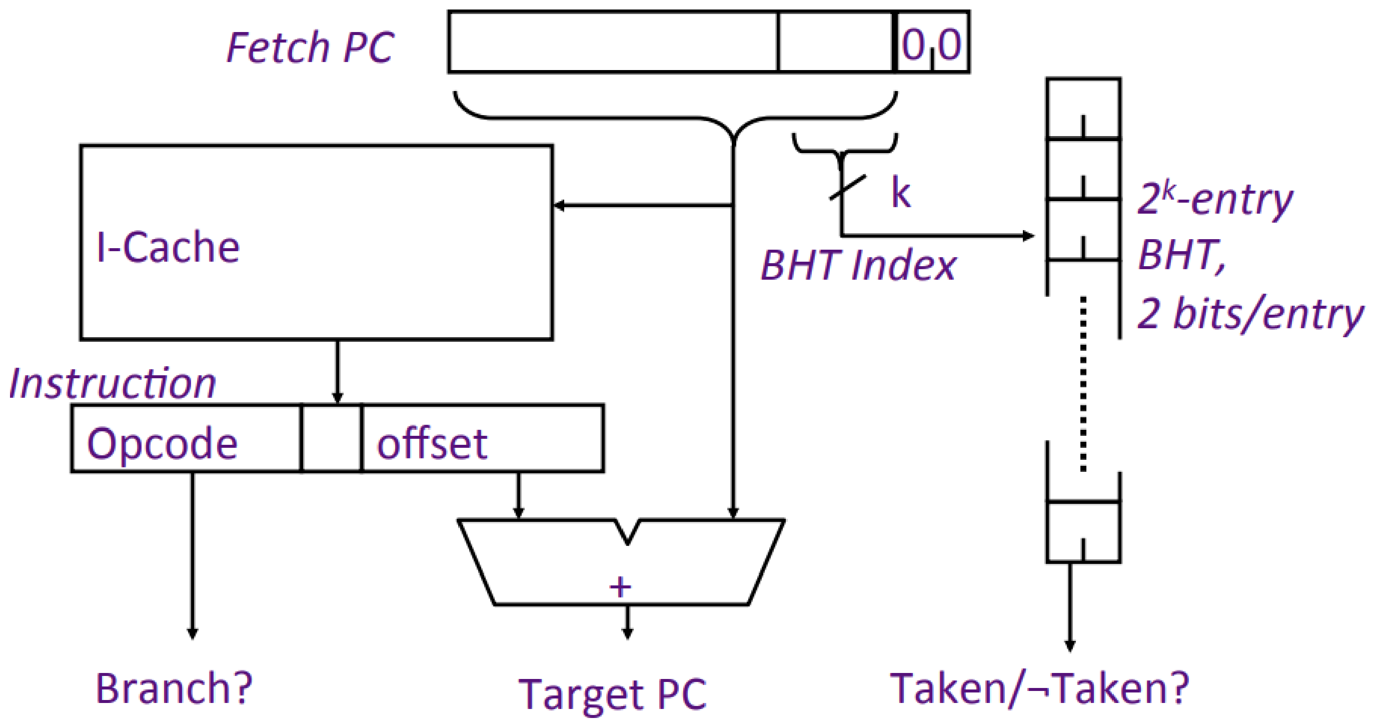


```
if (aa==2)
        aa=0;
if (bb==2)
        bb=0;
if (aa!=bb) {
```

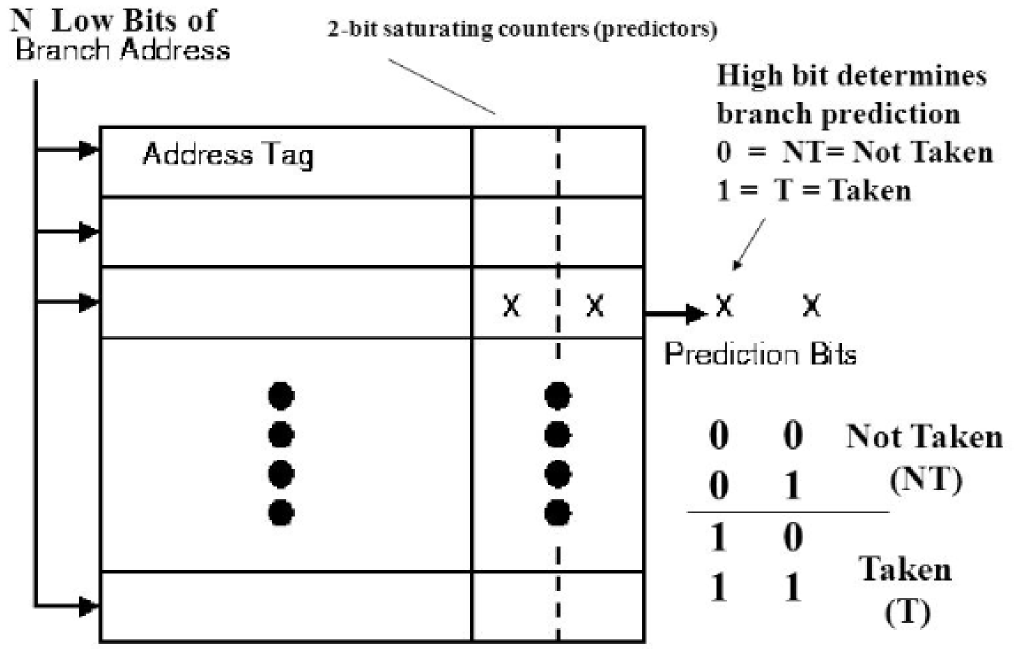
```
     addi  x3,x1,-2
     bnez  x3,L1        //branch b1   (aa!=2)
     add   x1,x0,x0     //aa=0
L1:  addi  x3,x2,-2
     bnez  x3,L2        //branch b2  (bb!=2)
     add   x2,x0,x0     //bb=0
L2:  sub   x3,x1,x2     //x3=aa-bb
     beqz  x3,L3        //branch b3   (aa==bb)
```

- Prediction can based other branches

Branch History Table

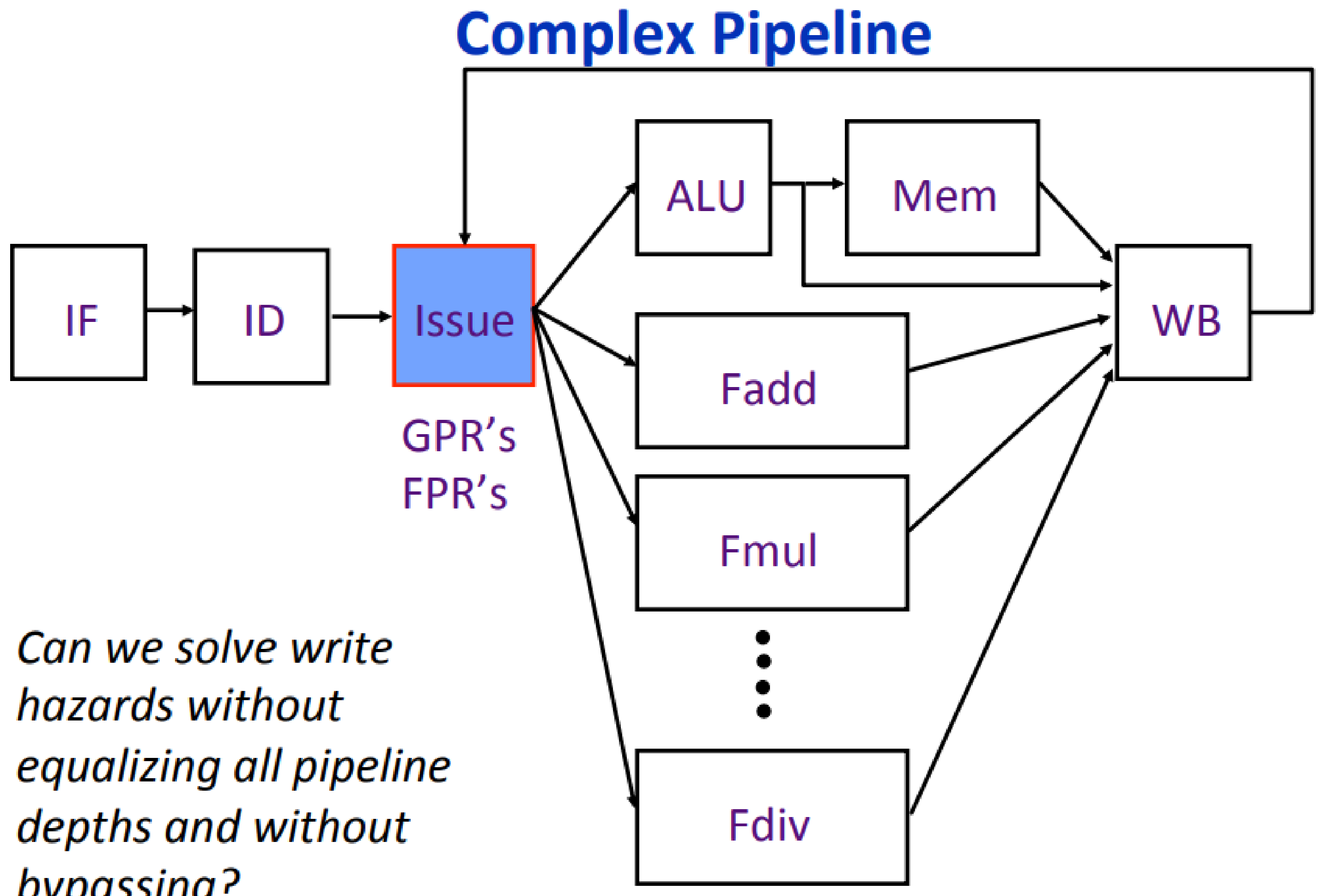


4K-entry BHT, 2 bits/entry, ~80-90% correct predictions


N Low Bits of Branch Address
2-bit saturating counters (predictors)
High bit determines branch prediction
0 = NT= Not Taken
1 = T = Taken
Address Tag
X X
X X
Prediction Bits
0 0 Not Taken
0 1 (NT)
1 0 Taken
1 1 (T)

The real world – Complex Pipeline

- Pipeline depths are not equalized in practice.
- Out-of-order scheduling



Can we solve write hazards without equalizing all pipeline depths and without bypassing?

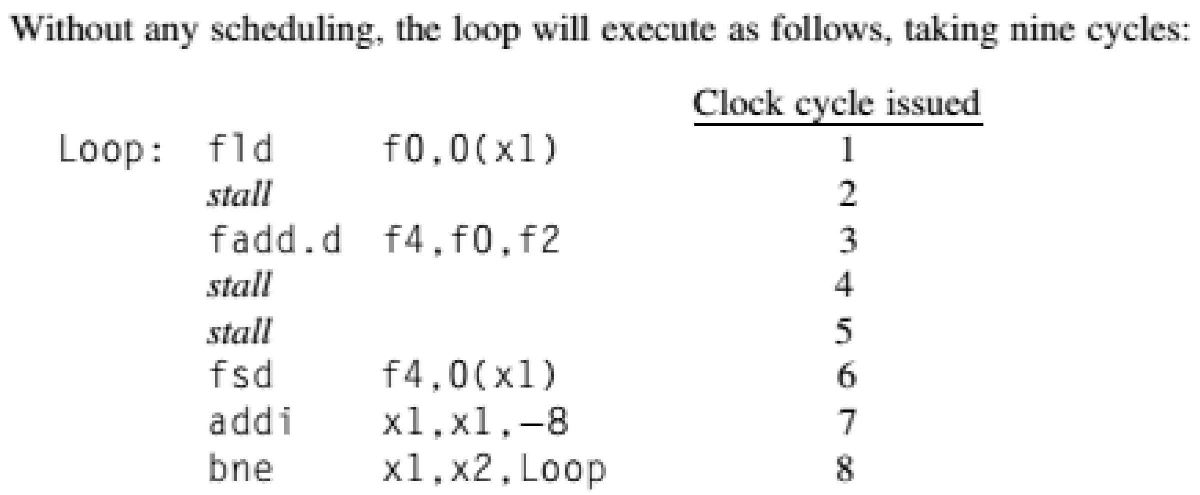
Loop Unrolling

- There are other methods to avoid branch stalls

```
Loop:  fld     f0,0(x1)      //f0=array element
       fadd.d  f4,f0,f2      //add scalar in f2
       fsd     f4,0(x1)      //store result
       addi    x1,x1,−8      //decrement pointer
                             //8 bytes (per DW)
       bne     x1,x2,Loop    //branch x1≠x2
```

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

How many stalls we need?

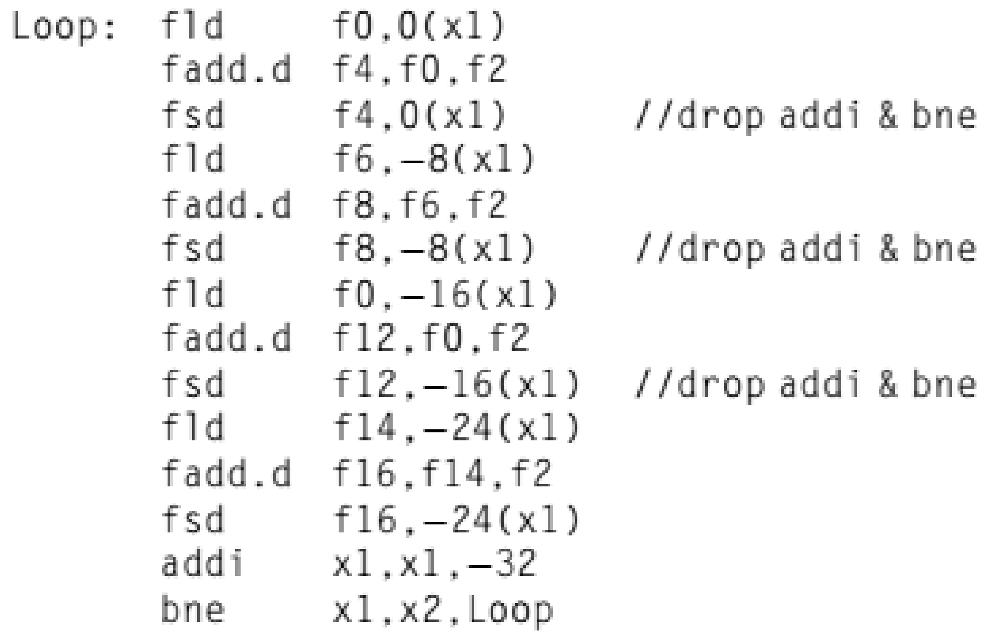


Without any scheduling, the loop will execute as follows, taking nine cycles:

```
                                      Clock cycle issued
Loop:  fld     f0,0(x1)                       1
       stall                                  2
       fadd.d  f4,f0,f2                       3
       stall                                  4
       stall                                  5
       fsd     f4,0(x1)                       6
       addi    x1,x1,-8                       7
       bne     x1,x2,Loop                     8
```

There are two more after branch instructions

Step 1 – Unrolling the loop

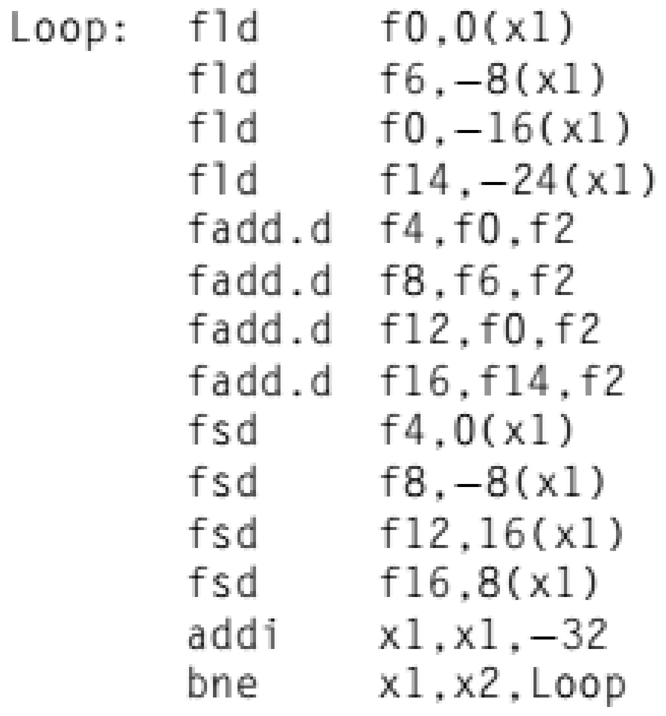


```
Loop:  fld     f0,0(x1)
       fadd.d  f4,f0,f2
       fsd     f4,0(x1)       //drop addi & bne
       fld     f6,-8(x1)
       fadd.d  f8,f6,f2
       fsd     f8,-8(x1)      //drop addi & bne
       fld     f0,-16(x1)
       fadd.d  f12,f0,f2
       fsd     f12,-16(x1)    //drop addi & bne
       fld     f14,-24(x1)
       fadd.d  f16,f14,f2
       fsd     f16,-24(x1)
       addi    x1,x1,-32
       bne     x1,x2,Loop
```

Reschedule the Code

Comparison (normalized to each branch)

- 11 cycles before unrolling
- 6.5 cycles after unrolling
  drop addi & bne instruction
- 3.5 cycles after rescheduling

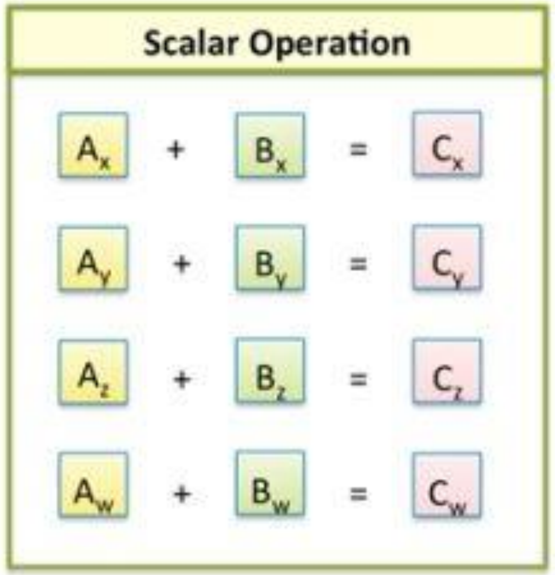


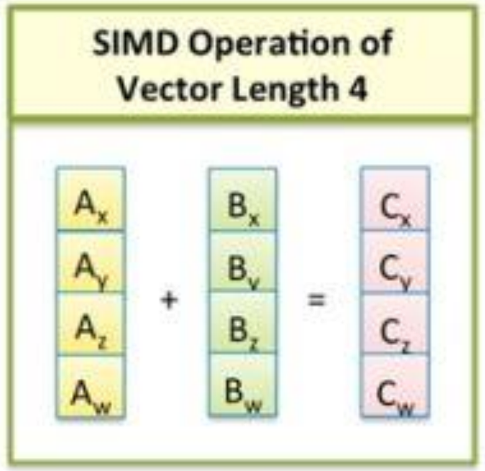
```
Loop:  fld     f0,0(x1)
       fld     f6,−8(x1)
       fld     f0,−16(x1)
       fld     f14,−24(x1)
       fadd.d  f4,f0,f2
       fadd.d  f8,f6,f2
       fadd.d  f12,f0,f2
       fadd.d  f16,f14,f2
       fsd     f4,0(x1)
       fsd     f8,−8(x1)
       fsd     f12,16(x1)
       fsd     f16,8(x1)
       addi    x1,x1,−32
       bne     x1,x2,Loop
```

AI Perspective – “Hard” Loop Unrolling

- Flynn Taxonomy
  - SISD: Single Instruction Single Data
  - SIMD: Single Instruction Multiple Data

- More Advanced Topics:
  - Multi-threading
  - SIMT -- GPU




SIMD Operation of Vector Length 4
A_x A_y A_z A_w + B_x B_y B_z B_w = C_x C_y C_z C_w

Intel® Architecture currently has SIMD operations of vector length 4, 8, 16

Conclusion

Key words: Pipeline, Hazards, Data Hazard, forwarding, stall, control hazard, branch prediction, loop unrolling, SIMD