Appendix C

Instructor: Josep Torrellas

CS433

Pipelining

- Multiple instructions are overlapped in execution
- Each is in a different stage
- Each stage is called “pipe stage or segment”
- Throughput: # inst completed/cycle
- Each step takes a machine cycle
- Want to balance the work in each stage
- Ideally:

  $$\text{Time per instruction} = \frac{\text{Time per inst in a non-pipelined}}{\text{\# pipe stages}}$$

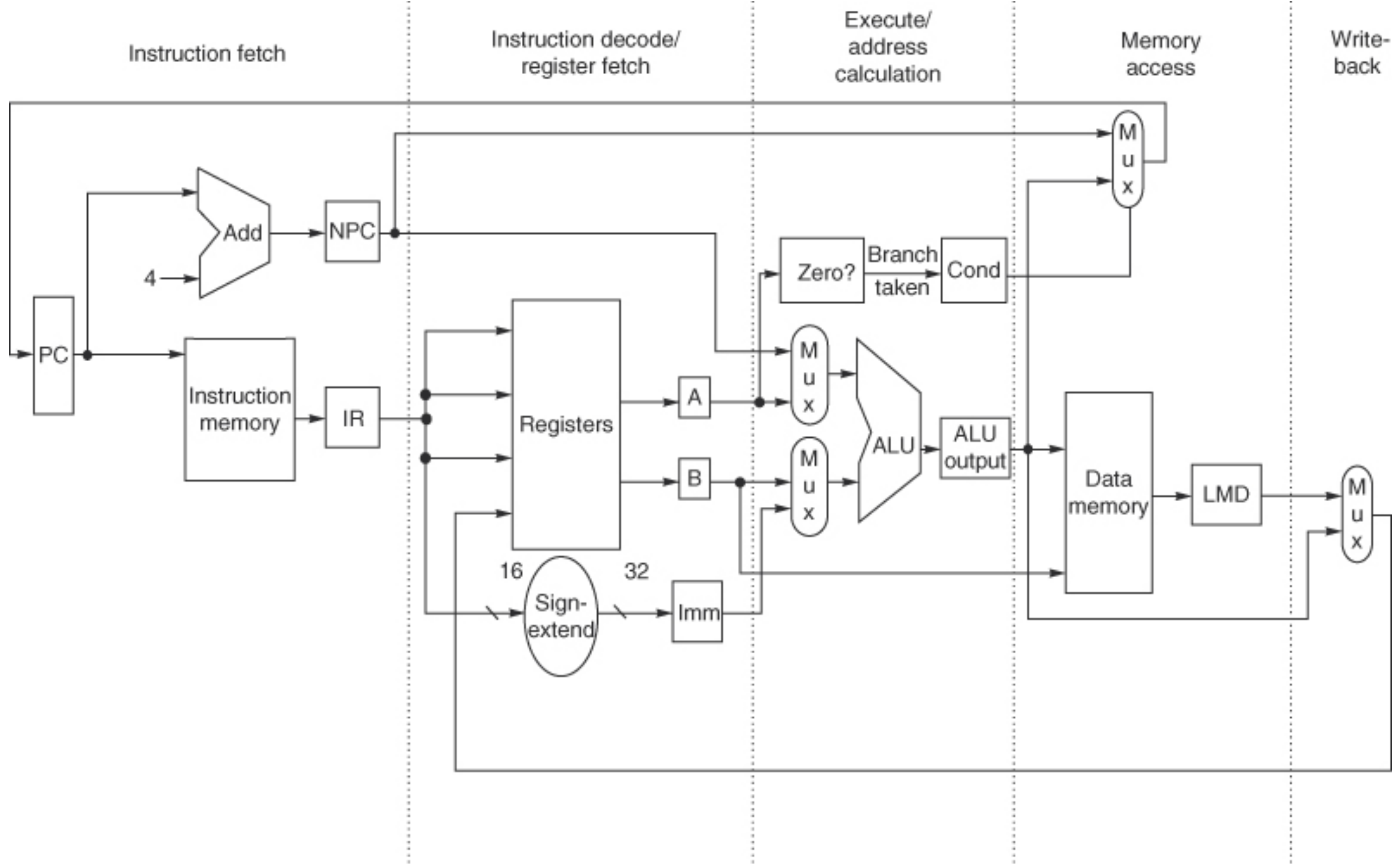

Instruction fetch
Instruction decode/ register fetch
Execute/ address calculation
Memory access
Write-back
PC
Add
4
NPC
Instruction memory
IR
Registers
A
B
16
Sign-extend
32
Imm
Zero?
Branch taken
Cond
Mux
ALU
ALU output
Data memory
LMD
Mux
Mux

Figure C.21

Implementation of RISC Instructions

1. Instruction Fetch cycle (IF)

   IR ← Mem[PC]   ; IR holds the instruction

   NPC ← PC+4

2. Instruction decode/register fetch cycle (ID)

   A ← Regs[rs]   ; decode the instruction

   B ← Regs[rt]   ; in the meantime

   Imm ← sign-extend imm field of IR   ;Regs A, B, Imm

   ; ok if some of this is not needed

3. Execution /Effective address cycle (EX)

- memory ref: ALU output ← A+Imm
- Reg-Reg (ALU op): ALU output ← A op B
- Reg-Immed (ALU op): ALU output ← A op Imm
- Branch: ALU output ← NPC+ (Imm << 2)

  ;address of target

  cond ← (A op O) ; op = equal,

  = not equal

/* note: no instructions need to do 2 of these operations */

/* note: Imm has word count for branches; need to shift by 2 to get bytes to add to PC */

4. Memory Access/Branch Completion Cycle (MEM)

/* only for LD,ST,BR */

- Memory access:

```
LMD ← Mem[ALU output]   ;for loads. Store data in
                          ; load mem data register
Mem[ALU output] ← B      ; for stores
```

- Branch

```
if (cond)
    PC ← ALU output
else
    PC ← NPC
```

5. Write-back cycle (WB)

- Reg-Reg ALU instr: Regs[rd] $\leftarrow$ ALU output
- Reg-Imm ALU instr: Regs[rt] $\leftarrow$ ALU output
- Load Instruction: Regs[rt] $\leftarrow$ LMD

Now we will try to pipeline it

We need: At the end of each cycle, the data is stored in some registers (PC,LMD,Imm,A,B,…). This allows other instructions to execute too.

- Branches $\rightarrow$ 4 cycles

  Rest of ins $\rightarrow$ 5 cycles

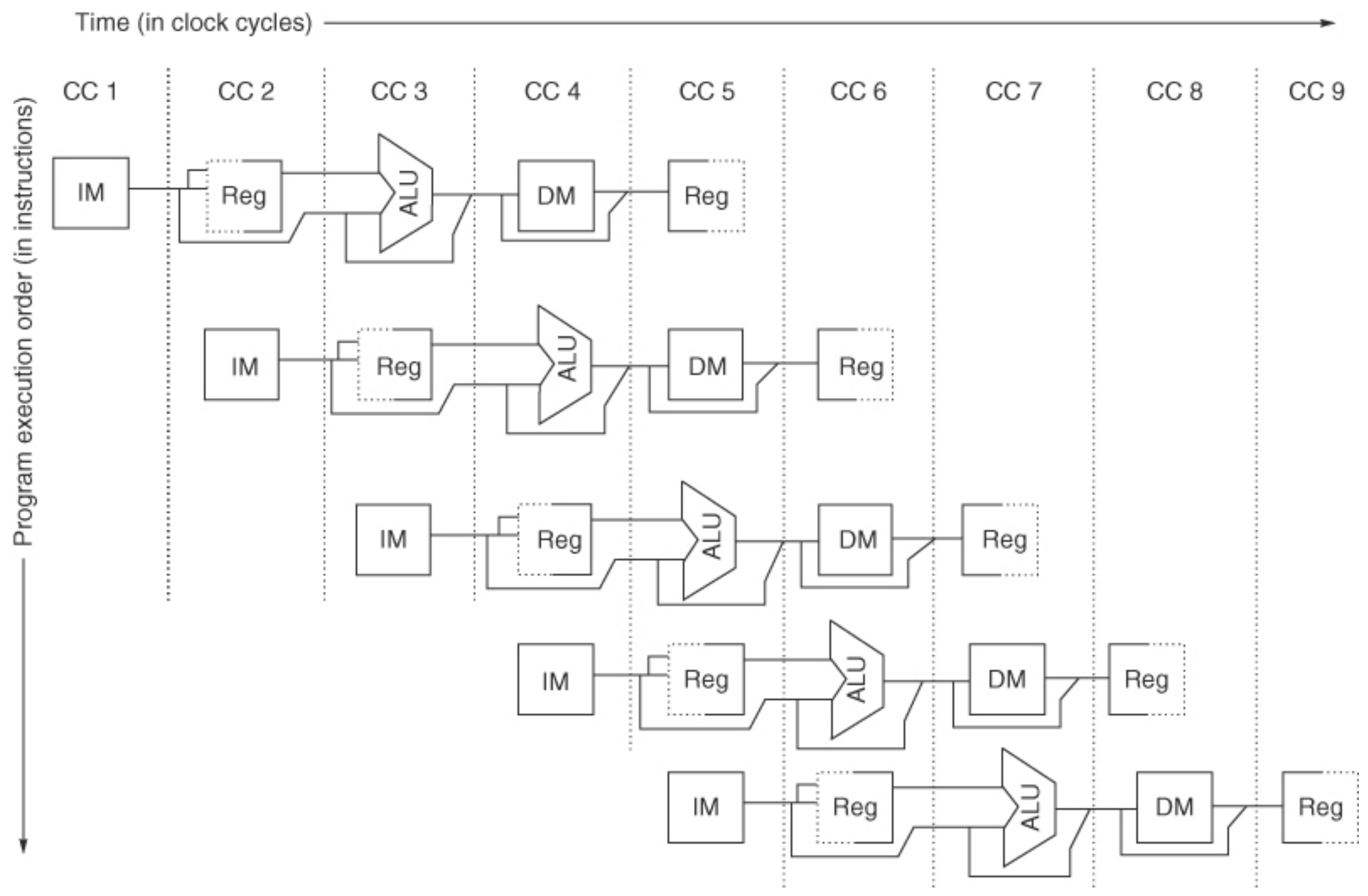


Figure C.1and C.2

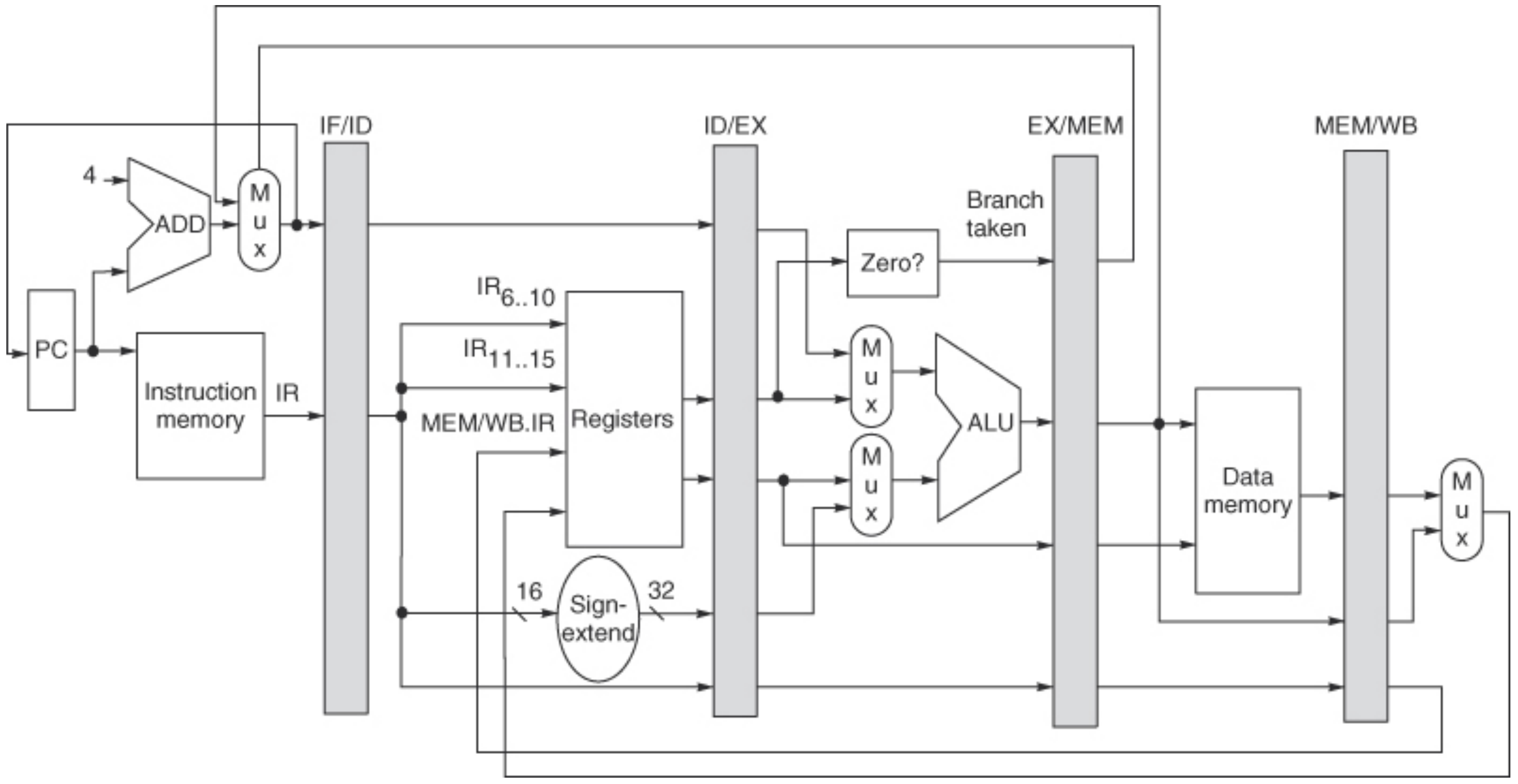


Figure C.22 and C.23

Why does it work?

- Use separate I and D caches
- Register file can be read/written in 0.5 cycles
- PC: incremented in IF

  if branch taken, in EX, add PC+ (Imm << 2)
- Cannot keep any state in IR → need to move it to another register every cycle → see picture

  These registers IF/ID, ID/EX, EX/MEM, MEM/WB subsume the temp ones

  e.g. Destination Reg in a LD

Control of the pipeline: set the control of the 4 MUXES (Figure C.22)

- ALU stage MUXES: set depending on instruction type which is set by ID/EX. IR
  - top one: branch or not
  - bottom one: reg-reg ALU or other
- MUX in IF:

  chooses between PC+4 and EX/MEM. ALUOutput

  controlled by EX/MEM.cond
- MUX in WB:

  controlled by whether inst. is a LD or an ALU op

- A final MUX shown in WB: chooses the field in IR that determines what reg to use to store the result
  - in reg-reg ALU MEM/WB. $IR_{16\ldots20}$ (rd)
  - in reg-imm ALU and LD MEM/WB. $IR_{11\ldots15}$ (rt)

Performance Issues

$$\begin{tikzcd}
 & \text{extra regs} \\
\text{Pipelining: } \rightarrow \text{ each instruction is slower} \arrow[ur] \arrow[dr] & \\
 & \text{time of slowest pipe stage}
\end{tikzcd}$$

$\rightarrow$ but throughput is higher

Example

Unpipelined: 10ns cycle time

4 cycles for ALU (40%), branch (20%)

5 cycles for mem (40%)

pipelining: adds 1 ns to clock

speedup in execution rate?

Unpipelined: avg inst = clock * avg CPI = 10*((40%+20%)*4 + 40%*5) = 44 ns

pipelined = 11 ns

Speedup= 44/11 = 4___________

Pipeline Hazards

Situations that prevent the next instruction from executing its designated clock cycle

→ Structural: resource conflicts e.g. not enough multipliers

→ Data: instruction depends on the result of a previous one. e.g. ADD <u>R1</u>, R2, R3

ADD R4, R5, <u>R1</u>

→ Control: results from instructions that change the PC. e.g. BEQ R1, label

ADD R7, R6, R7

As a result, the pipeline may have to stall

Note : In Hazards (and cache misses) : instruction before hazard continue

after hazard stall

$$\text{Speed up From Piplining} = \frac{\text{Avg. inst. Time unpipelined}}{\text{Avg. inst. Time pipelined}} = \frac{\text{CPI Unpipelined} * \text{Clock cycle unpipelined}}{\text{CPI Pipelined} * \text{Clock cycle pipelined}}$$

$$\text{CPI}_{\text{pip}} = \text{Ideal CPI} + \text{Pipeline stall clock cycles per inst.}$$

$$\text{Speedup} = \frac{\text{Pipeline Depth}}{1+ \text{Pipeline stall cycles per inst.}}$$

Structural Hazards

- Some Combination of inst. Cannot be accomodated because of resource conflicts
- Usually because some functional unit is not pipelined two instructions using it cannot proceed back to back
- Some resource has not been replicated enough

  Eg 1 register file port

  Combined I,D memory

Result : Pipeline stall, like if we had inserted a bubble.

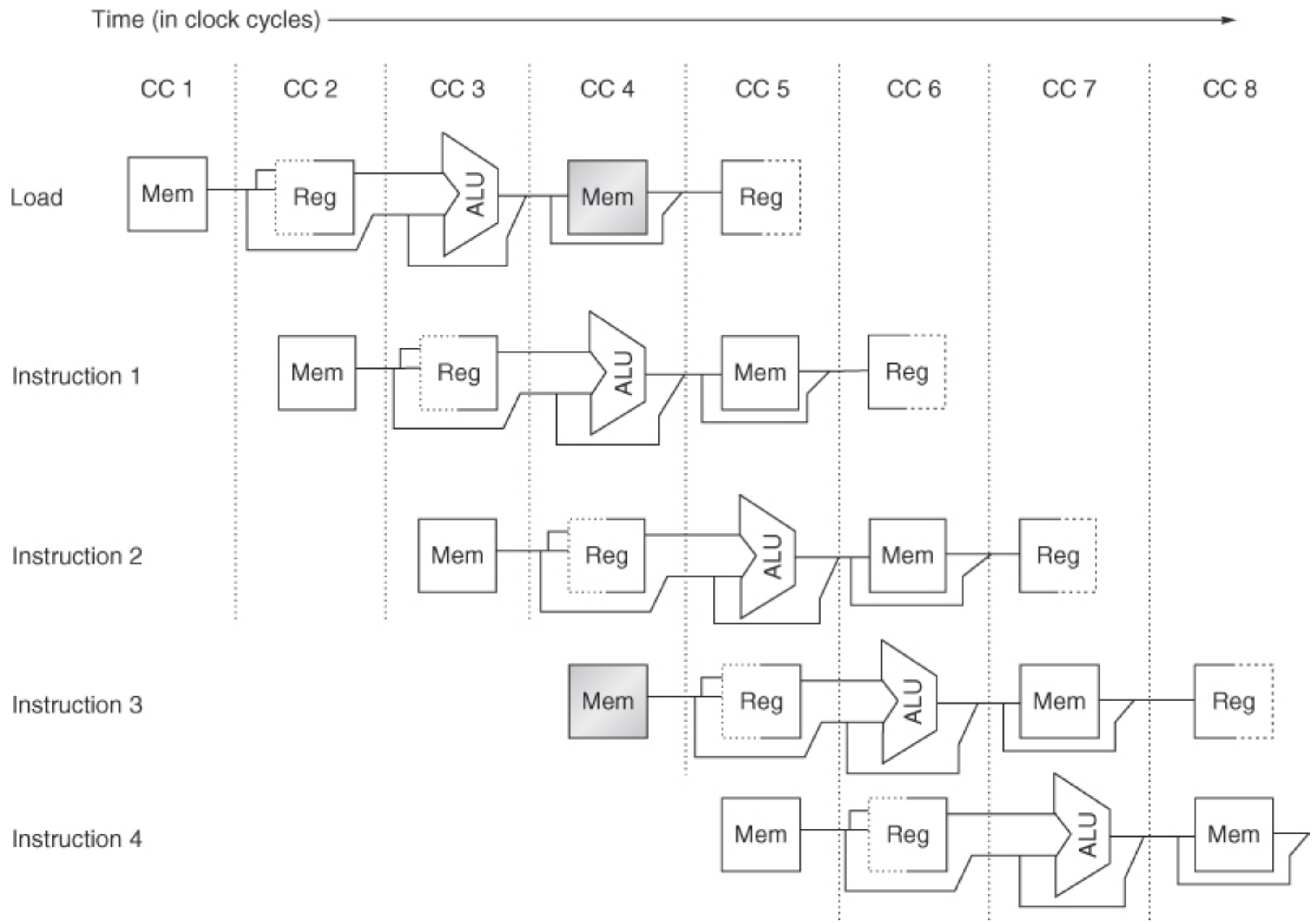


Figure C.4 and Figure C.5

Example : Machine 1 separate I,D

Machine 2: Unified I,D clock rate 1.05 higher

40% of instructions are data Accesses

Which is faster?

$$(\text{Avg. inst. time})_1 = \text{CPI} * (\text{Clock cycle time}) = 1 * (\text{Clock cycle time})$$

$$(\text{Avg. inst. time})_2 = \text{CPI} * \frac{\text{Clock Cycle Time}}{1.05} = (1 + 0.4*1) * \frac{\text{Clock Cycle Time}}{1.05}$$

$$= 1.3 * (\text{Clock Cycle time})$$

- Why allow structural hazards ?
  - Reduce cost
  - speed up FUnit

Data Hazards

Occurs because pipelining changes the order of read/write accesses to operands

```
 1 ADD   R1, R2, R3
2 SUB    R4,R5,R1
3 AND   R6,R1,R7
4 OR      R8,R1,R9
5 XOR   R10,R1,R11
```

Left to their own devices all these instructions produce wrong results

To fix problem

4   Split register access W,R

2,3 Forwarding (bypassing or short circuiting).

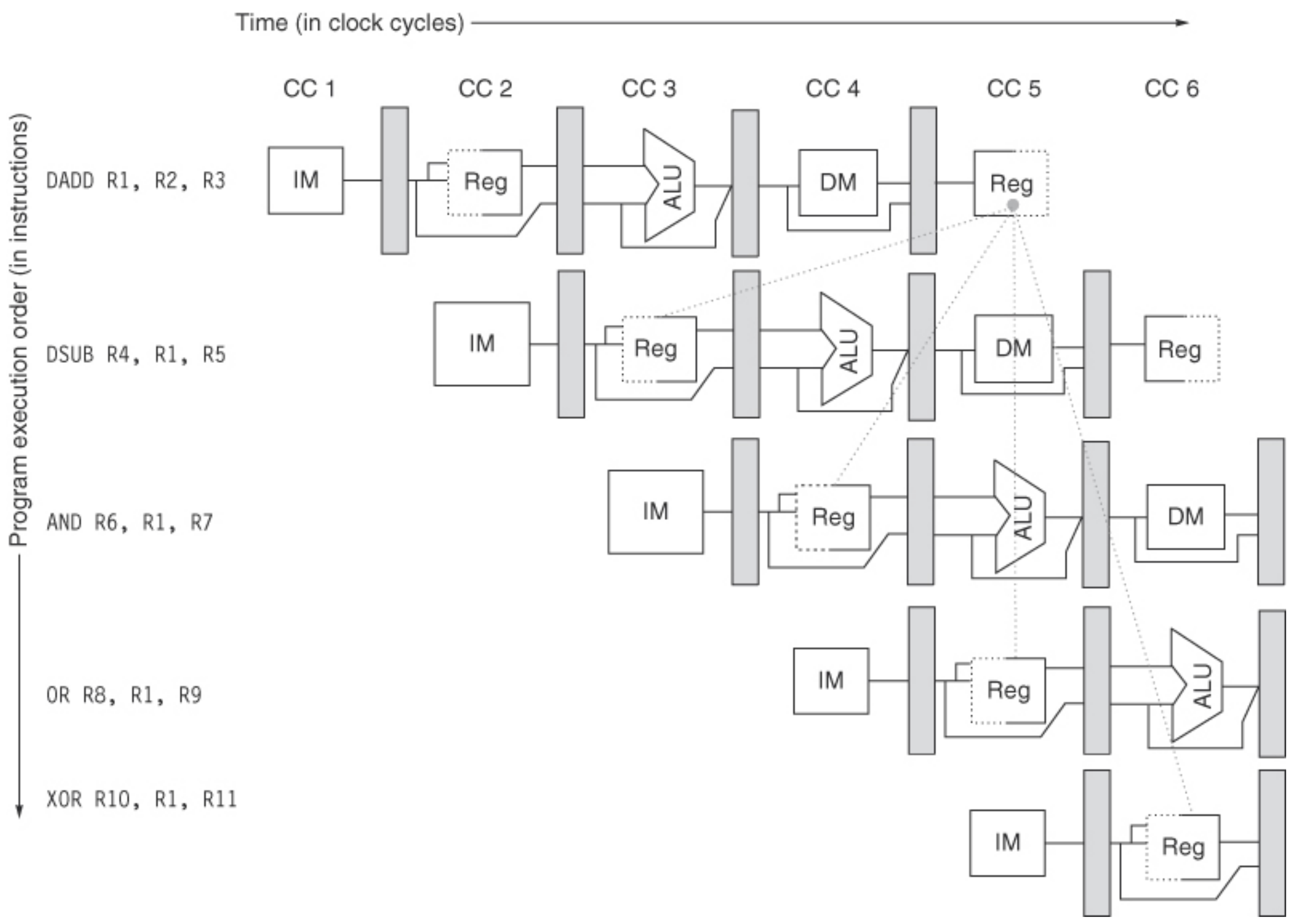

Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
Program execution order (in instructions)
DADD R1, R2, R3
DSUB R4, R1, R5
AND R6, R1, R7
OR R8, R1, R9
XOR R10, R1, R11
IM
Reg
ALU
DM
Reg

Figure C.6

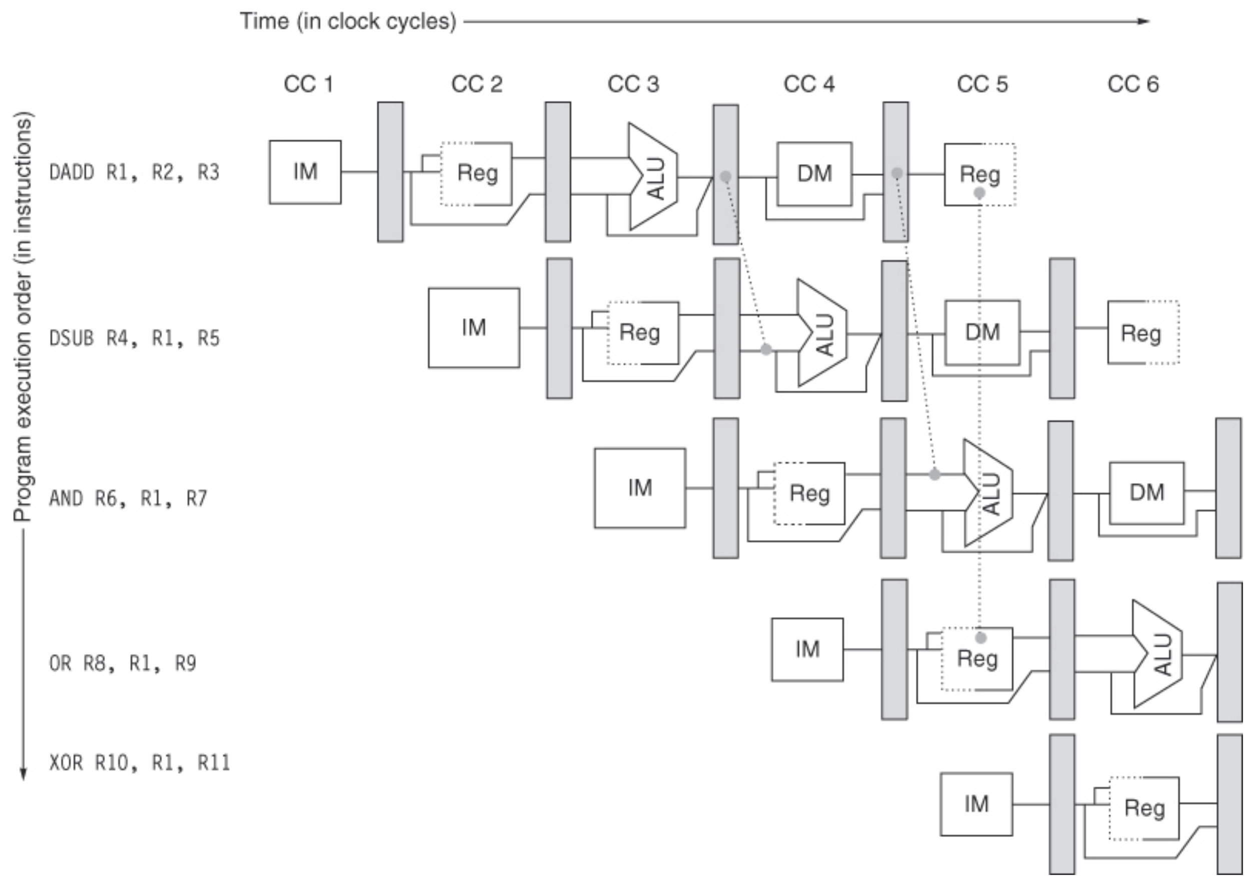

Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
Program execution order (in instructions)
DADD R1, R2, R3
DSUB R4, R1, R5
AND R6, R1, R7
OR R8, R1, R9
XOR R10, R1, R11
IM
Reg
ALU
DM

Figure C.7

Forwarding

- ALU result from the EX/Mem reg is fed back to the ALU input latches
- If forwarding hardware detects the dependency , it selects the forwarded value instead of value from the reg.
- Do this also between instructions that are 1 instruction apart
- Need forwarding path to the data memory input

```
 ADD R1 , R2 , R3
 LW   R4 , 0(R1)
 SW  12(R1) , R4
```

- Overall : No Cycle lost (for now)   --- One exception…

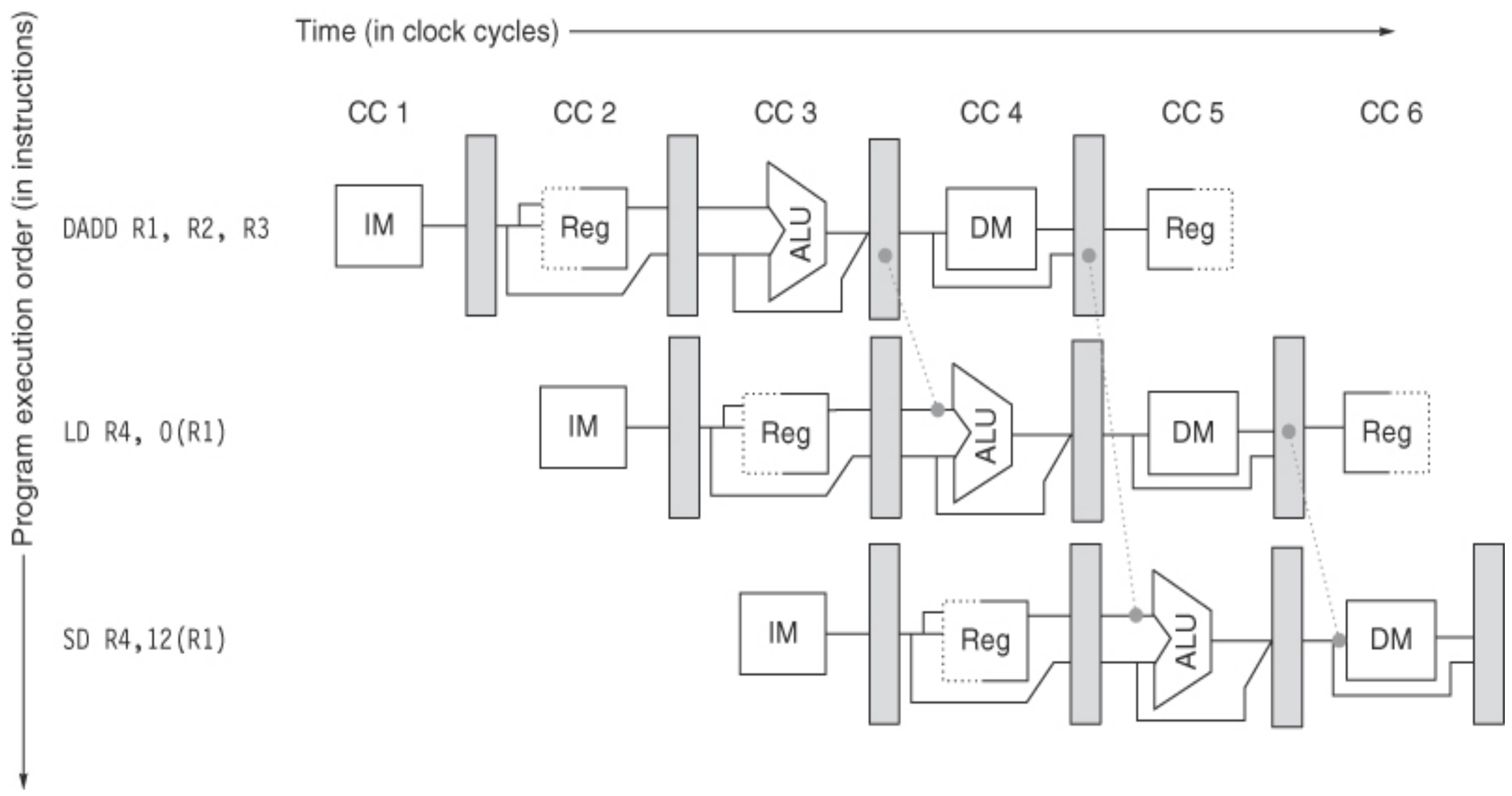

Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
Program execution order (in instructions)
DADD R1, R2, R3
LD R4, 0(R1)
SD R4,12(R1)
IM
Reg
ALU
DM
Reg

Figure C.8

Classifying Data Hazards

- RAW(Read after Write) : i + 1 tries to read before i writes

```
ADD  R1
ADD  R7 , R1
```

- WAW(Write after Write) : i + 1 tries to write before i writes
  - Not Possible in MIPS
  - Could happen if ALU instr. Wrote in the MEM stage and data mem accesses took 2 pipe stages

```
LW  R1,0(R2)    IF ID EX MEM1 MEM2 WB
ADD R1,R2,R3      IF  ID  EX       WB
```

- WAR(Write after Read) : i + 1 tries to write before i reads
  - Not possible in MIPS because inst. read first ID, write in WB
  - Occurs when some inst write early
    read late (for example autoincrement addressing)

```
SW     R1,0(R2)+   IF ID EX MEM1 MEM2 AU WB    autoincrement
ADD  R2, R3, R4       IF  ID  EX       WB
```

- RAR(Read after Read) : No Hazard

Data Hazards that require stall - loads

```
LW   R1 , O(R2)
SUB  R4 , R1 , R5
AND R6 , R1 , R7
 OR   R8, R1 , R9
```

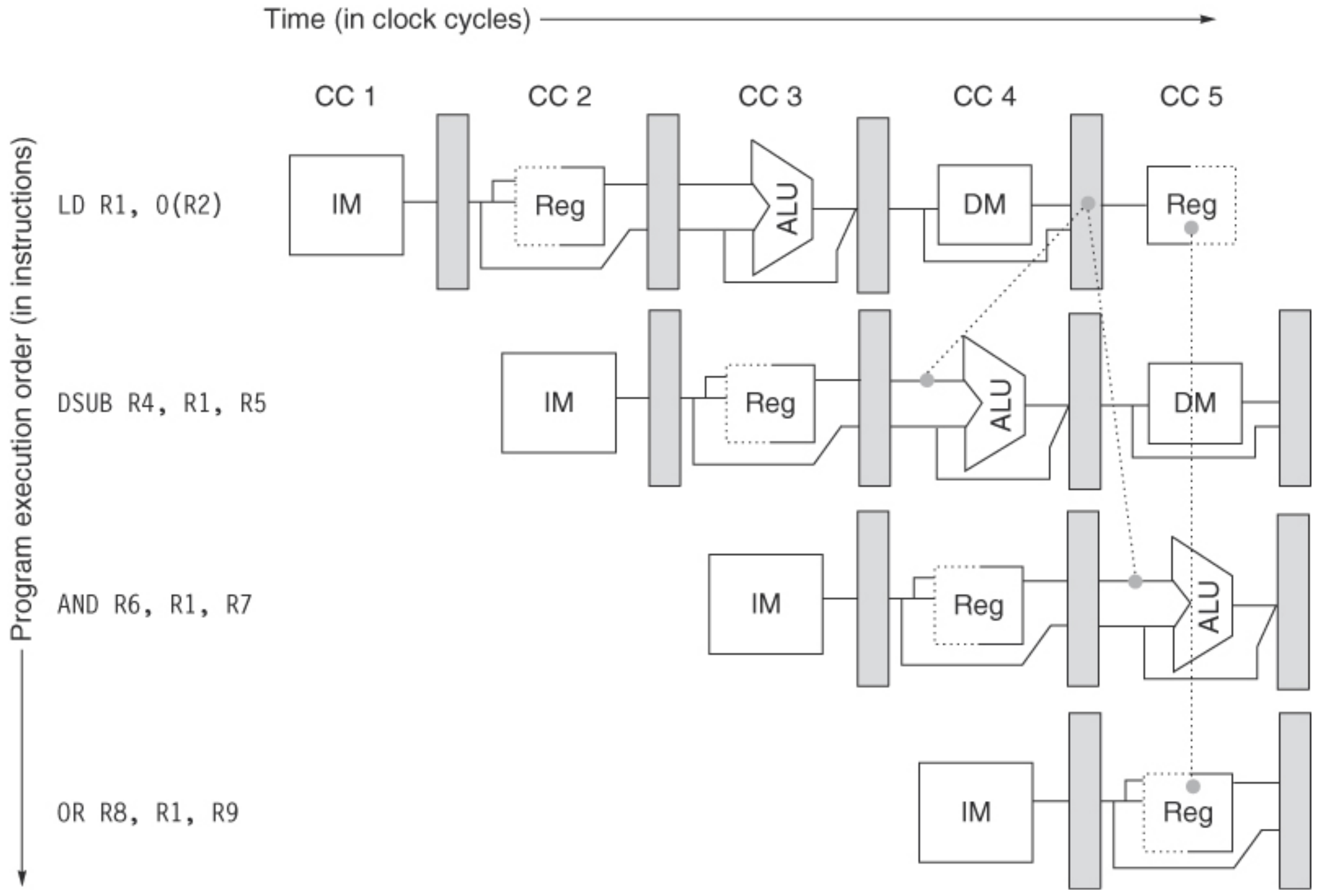


Figure C.9 and C.10

How to handle these hazards

1 Add hardware(pipeline interlock) to detect hazard and stall then pipeline until the hazard is cleared
   - The CPI of the SUB instruction increases by 1

2 Pipeline scheduling by the compiler : avoid putting a load followed by immediate use of the load register

```
a = b + c          lw   Rb  , b
d = e - f          lw   Rc  , c
                   lw   Re  , e
                   add Ra  , Rb  , Rc
                   lw   Rf  , f
                   sw  a , Ra
                   sub Rd  , Re  , Rf
                   sw  d , Rd
```

   - Pipeline schedule can increase the reg. count required
   - It is easier if scheduling happens within Basic Blocks: A basic block is a straight-line code sequence with no transfers in or out, except at the beginning or end

Control Issues

- Moving an instruction from ID->EX: issue it
- For MIPS , all data hazards can be detected in ID.
  If hazard exists -> don’t issue
- ID also determines what forwarding will be necessary
- When hazard is detected : -> insert a pipeline stall and prevent
  the instructions in IF or ID from advancing

See Figure C.25

This requires :

1. Change the control portion of ID/EX to all zeros -> NOP (does nothing)
2. Recirculate the contents of the IF/ID reg to hold the stalled instruction

For forwarding it is similar :

Compare EX/MEM.IR[rd] ==ID/EX.IR[rs]

Figure C.26

It is also necessary to enlarge the data path + add multiplexors

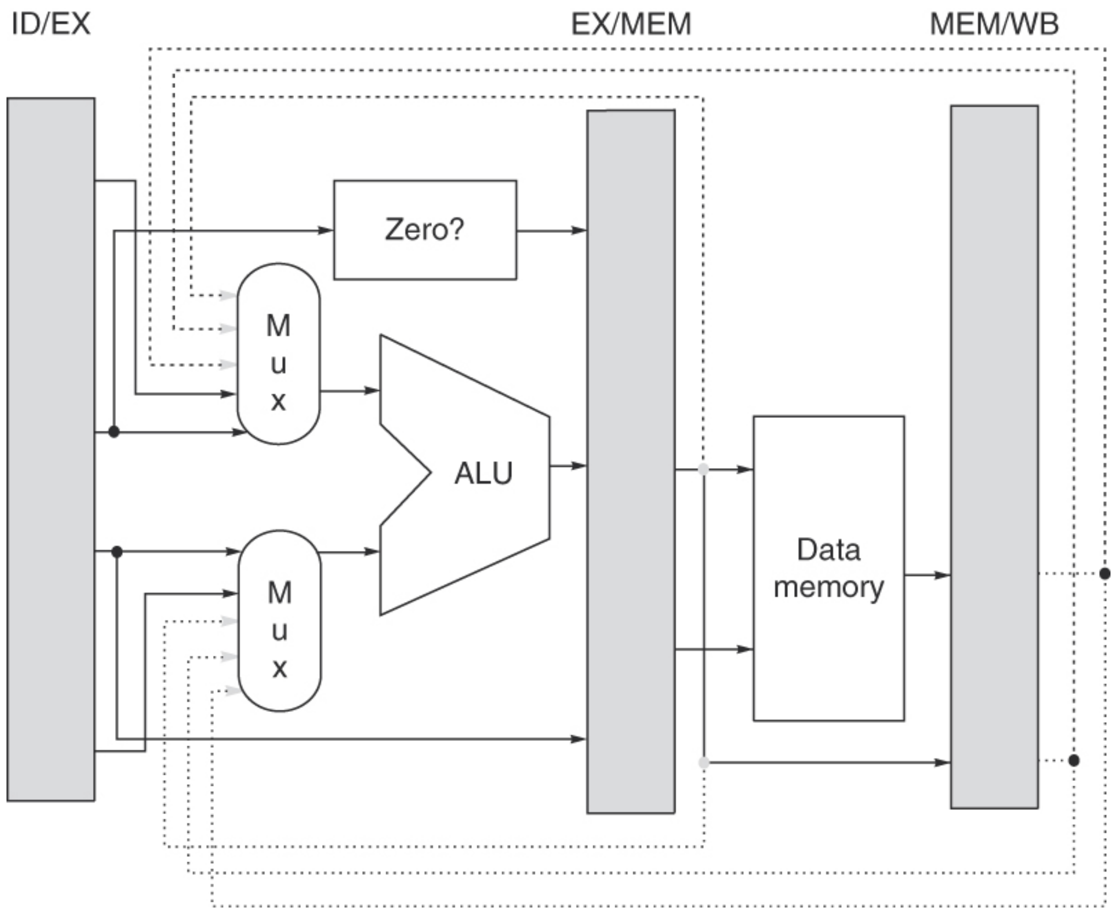


Figure C.27

Control Hazards: Branches

- When a branch is executed , it may or may not be taken
- If taken , the PC is not changed (usually) until
  the end of EX-> end of address calculation
- Easiest way : Stall pipeline as soon as the branch is detected until the EX
- Need to repeat the IF of the instruction following the branch (not necessary if not taken)
- Overall two cycles lost

Reducing Branch Stalls

- Do, as soon as possible :
  - Find out whether or not the BR is taken
  - Find out the target addr.

- How ?

  - Complete the testing for zero or not zero (BEQZ, QNEQZ) by the end of ID cycle (instead of EX)
    -> move the zero test to ID
  - Compute the target in the ID (instead of EX)
    -> requires extra adder
    -> therefore : only 1 clock cycle stall (Branch delay)

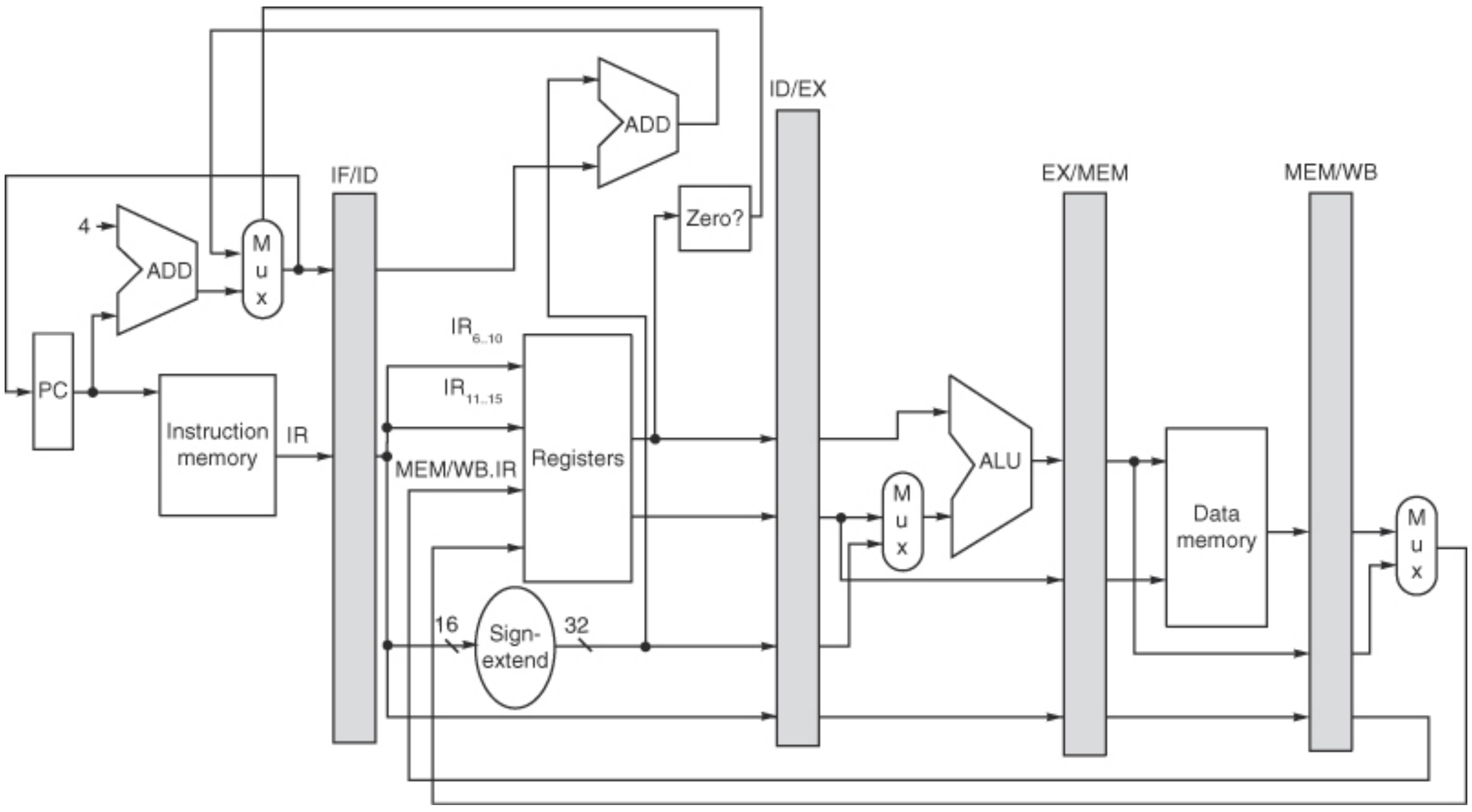


Figure C.11 and C.28

Branch Behavior of pgms.

- Conditionals dominate
- Forward dominate
- 67% of conditional br. are taken!
- Usually, backward branches are taken more often than forward ones (loops)

Reducing Branch Penalties (Static Methods)

Static methods : compiler decides

1. Freeze pipeline until branch is resolved => simple (see prev. figure)

2. Predict branch not taken :

   -> continue as if nothing happens

   -> following instr. do not change state until BR is resolved

   -> if fail : flush pipeline

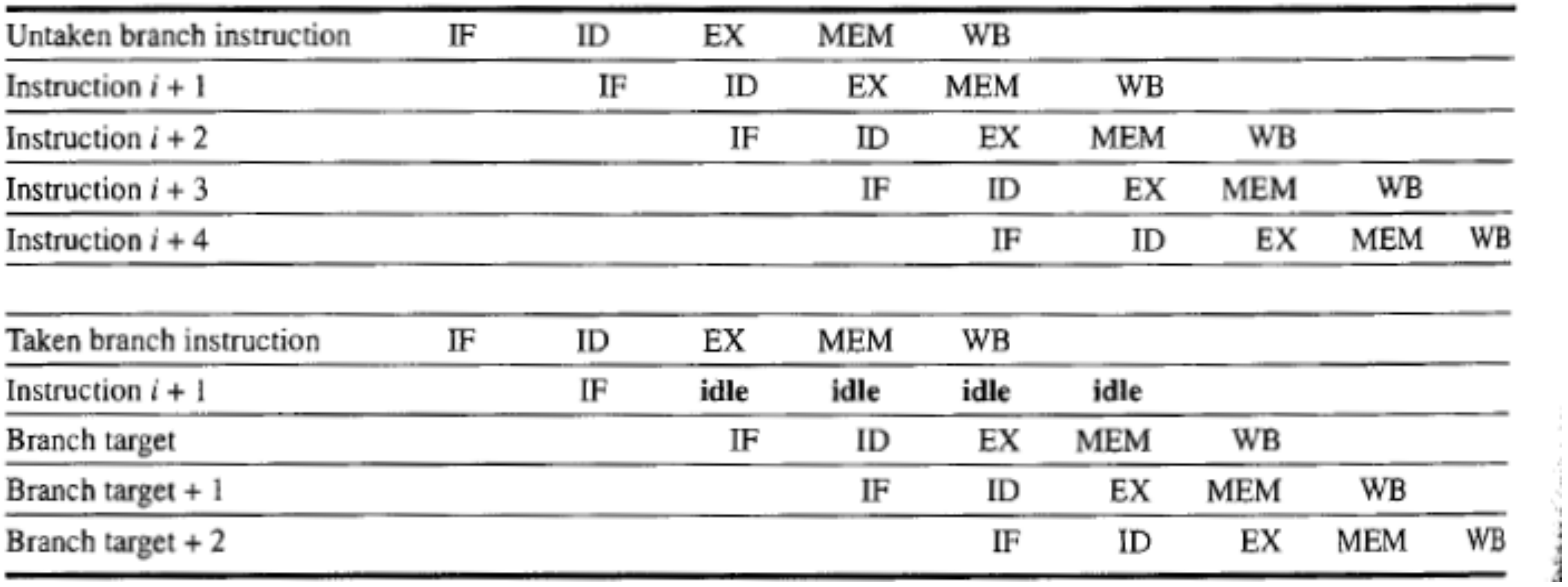
3. Predict taken

   -> after the decode + target address computed ->

   fetch from target (no advantage in MIPS)

Predict non-taken

Figure C.12



Untaken branch instruction	IF	ID	EX	MEM	WB				
Instruction *i* + 1		IF	ID	EX	MEM	WB			
Instruction *i* + 2			IF	ID	EX	MEM	WB		
Instruction *i* + 3				IF	ID	EX	MEM	WB	
Instruction *i* + 4					IF	ID	EX	MEM	WB

Taken branch instruction	IF	ID	EX	MEM	WB				
Instruction *i* + 1		IF	**idle**	**idle**	**idle**	**idle**			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

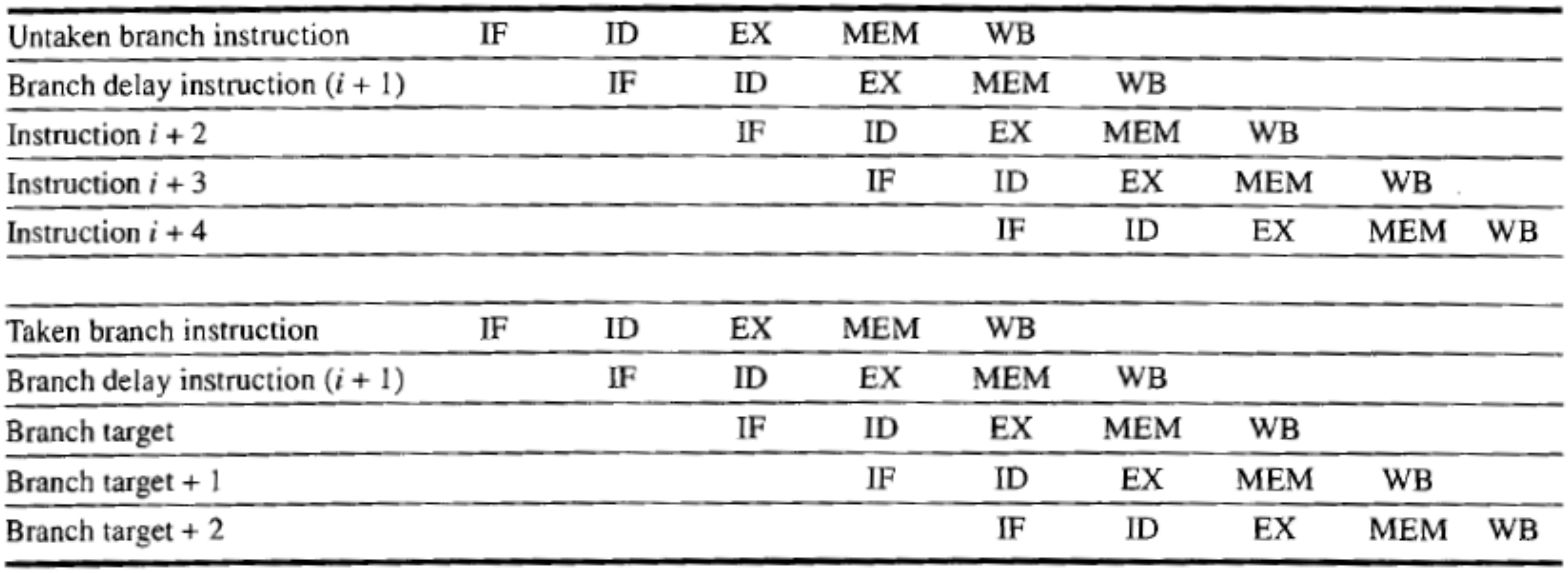
4. Delayed branch

-> the instruction(s) in the branch delay slot(s)
following the br will be executed irrespective
of the outcome of the branch.

-> MIPS 1 delay slot.

-> job of compiler: successor instruction(s) are valid and useful.

Figure C.13



Untaken branch instruction	IF	ID	EX	MEM	WB				
Branch delay instruction (*i* + 1)		IF	ID	EX	MEM	WB			
Instruction *i* + 2			IF	ID	EX	MEM	WB		
Instruction *i* + 3				IF	ID	EX	MEM	WB	
Instruction *i* + 4					IF	ID	EX	MEM	WB

Taken branch instruction	IF	ID	EX	MEM	WB				
Branch delay instruction (*i* + 1)		IF	ID	EX	MEM	WB			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

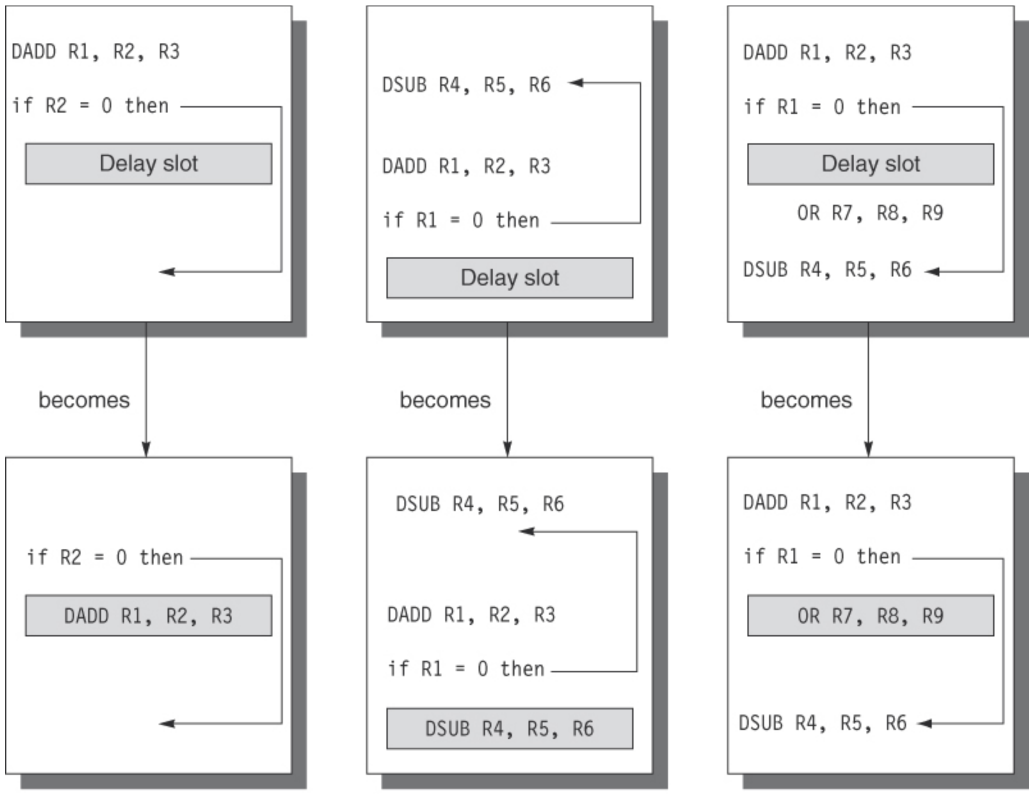
```
I1
BR R1   L
ADD R1
ADD R2

L: SUB R7
   SUB R8
```

```
BR R1   L
I1
ADD R1
ADD R2

L: SUB R7
   SUB R8
```

```
I1
BR R1   L
NOP
ADD R1
ADD R2

L: SUB R7
   SUB R8
```

```
I1
BR R1   L
ADD R1
ADD R2

L: SUB R7
   SUB R8
```

```
I1
BR R1   L2
SUB R7
ADD R1
ADD R2

L: SUB R7
L2: SUB R8
```



(a) From before (b) From target (c) From fall-through

Figure C.14

Delayed Branches (more)

- Cannot put a branch in a delay slot
  - Since longer branch delays in newer machines
    -> schemes for branch prediction (R4000 = 3 cycles)
- With 1 delay slot : need to have an extra PC
  - /* in case an interrupt occurs in the instruction
    in the branch delay slot */
- Impact on the ideal CPI :

$$\text{pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{branch freq.} * \text{branch penalty}}$$

Example : R4000

Branch Scheme	Penalty unconditional	Penalty untaken	Penalty taken
Stall pipeline	2	3	3
predict taken	2	3	3
predict untaken	2	0	3
if frequency ->	4%	10%	6%

$$CPI_{stall} = 1 + 0.04*2 + 0.10 * 3 + 0.06*3 = 1.56$$

How to Statically predict Branches

1 Examination of program behavior

- Since most branches taken → always predict taken
- Since bck branches taken → predict bck taken , fwd not taken

2 Use profiles of previous runs

Why ? <u>Branches are usually taken</u> or <u>not-taken</u>

bimodal distibution

Overall : CPI from pipelining for Integer spec 92 programs

$$CPI_{pip} = 1 + \underbrace{\underline{0.06}}_{\text{from branch}} + \underbrace{\underline{0.05}}_{\text{from load}} = 1.11$$

Handling Multicycle Operations

- Floating point operations usually take several EX cycles
- There are several floating point functional units
- Example : Assume we have 4 separate FU
  - Integer unit : Ld,St, integer ALU , Br
  - FL Pt and integer MPY
  - Fl Pt adder
  - Fl Pt and integer div

assumptions : 1 Not pipelined

2 If instruction i cannot proceed to EX,
entire pipeline behind stalls

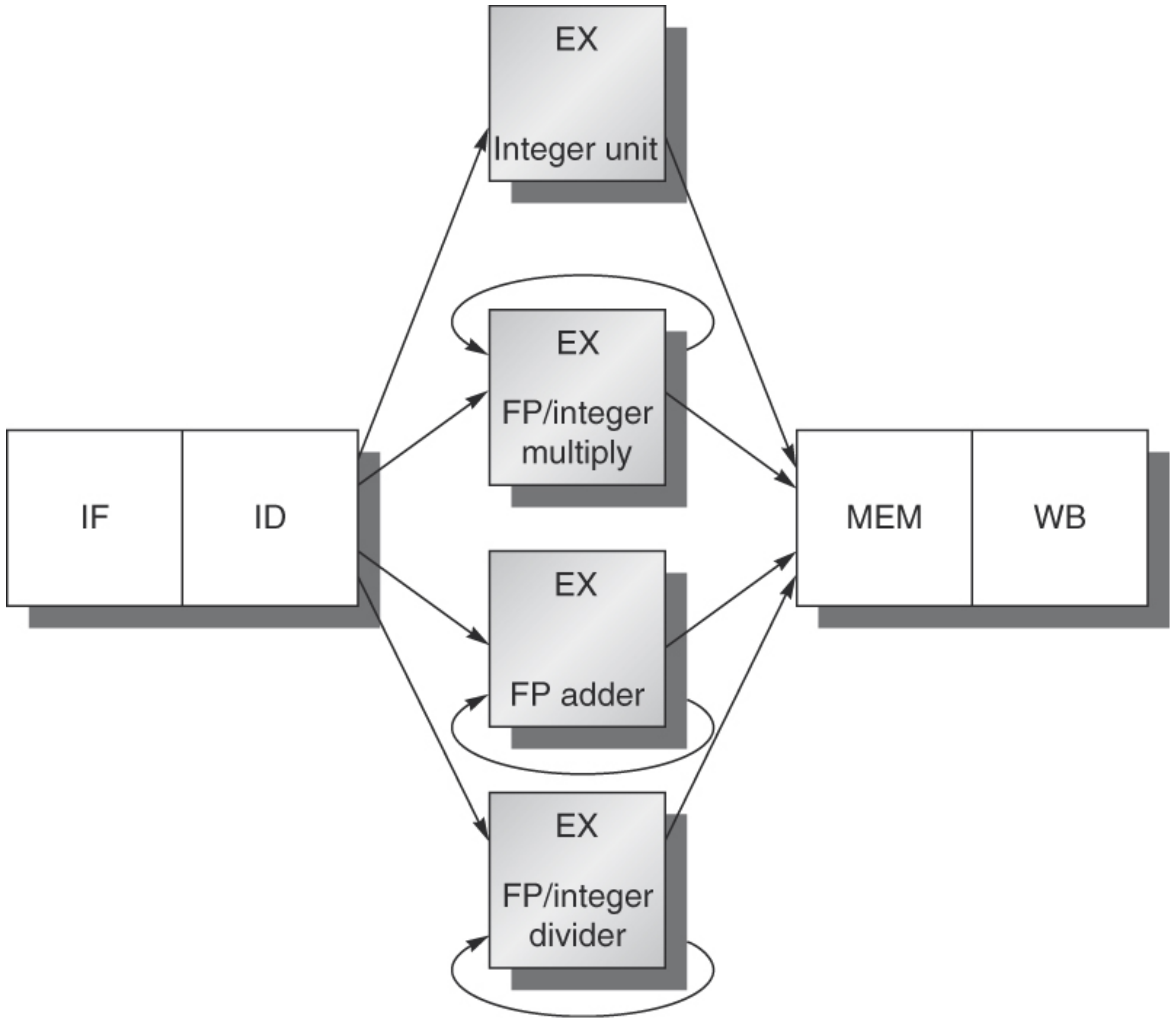

EX
Integer unit
EX
FP/integer multiply
IF
ID
EX
FP adder
EX
FP/integer divider
MEM
WB

Figure C.33

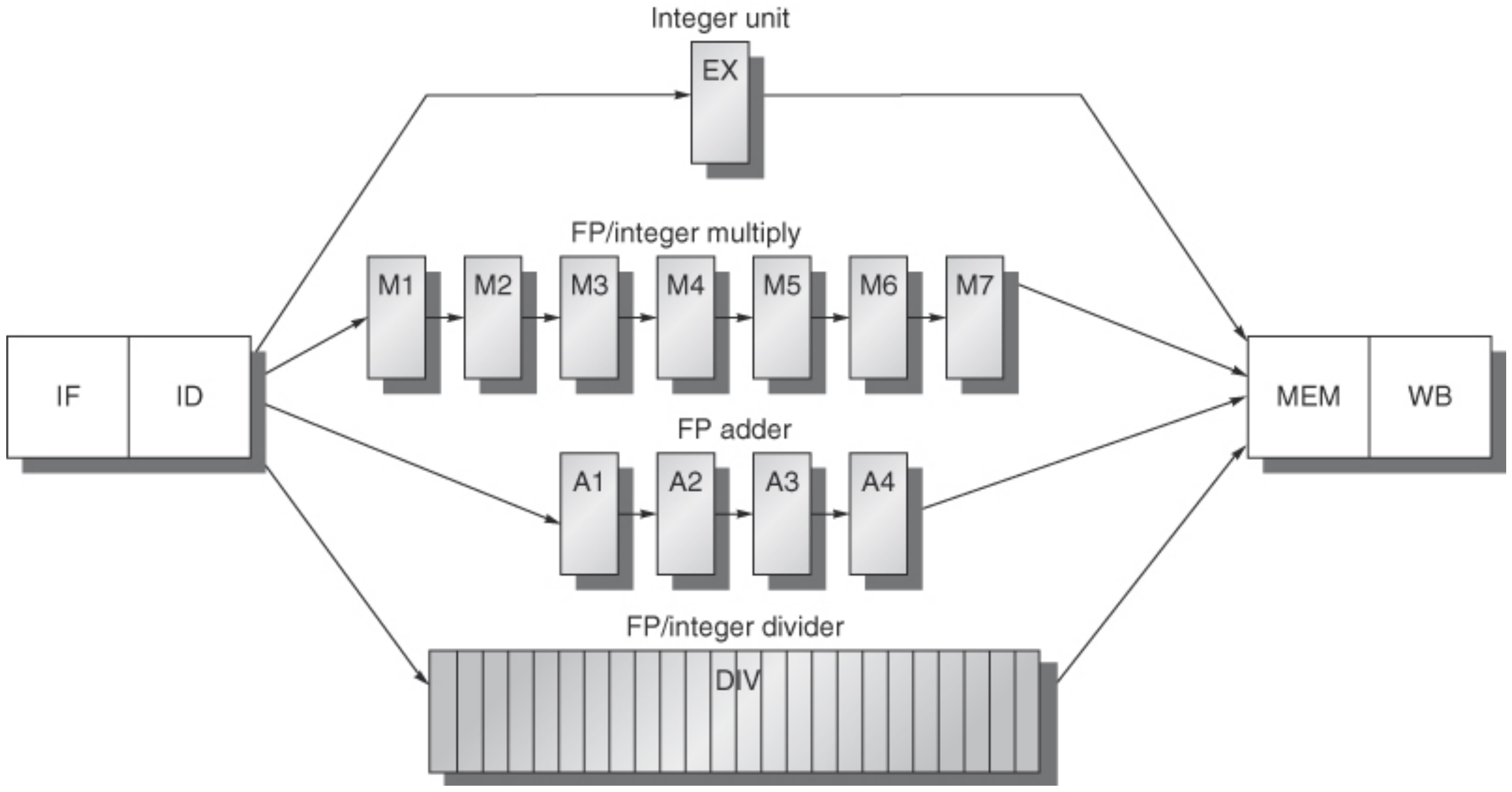

Integer unit
EX
FP/integer multiply
M1
M2
M3
M4
M5
M6
M7
IF
ID
MEM
WB
FP adder
A1
A2
A3
A4
FP/integer divider
DIV

Figure C.35

Resulting Pipeline

For independent instructions

MULTD	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
ADDD		IF	ID	A1	A2	A3	A4	MEM	WB		
LD			IF	ID	EX	MEM	WB				
SD				IF	ID	EX	MEM	WB			

- Obviously , for the pipelined FU , need extra regs

Problems

1. Divide FU is not pipelined → may suffer structural hazards
2. Because instructions have varying running times, we may try several WB in same cycle
3. Because instructions do not reach WB in order → WAW possible
   WAR not possible : reg always reads in ID

4. Instructions can complete out of order → pbms w/exceptions

5. Long latency operations → frequent RAW stalls

```
     MULTD    IF  ID  M1  M2  M3  M4  M5  M6  M7  MEM  WB
  {  F0,F4,F6
5 {  ADD          IF  ID  --  --  --  --  --  --  A1   A2   A3   A4   MEM
  {  F2,F0,F8
     SD,F2,6(R2)      IF  --  --  --  --  --  --  ID   EX   --   --   MEM

     MULTD    IF  ID  M1  M2  M3  M4  M5  M6  M7  MEM  WB
  {  F0,F4,F6
2 {               IF
  {                   IF
     ADD F2,F4,F6         IF  ID  A1  A2  A3  A4  MEM  WB
                                                  Only 1 write port
                                                   in Reg. File
```

How to Prevent Multiple WB?

- Each instruction, in the ID stage, tracks the current use of the write port of the register file. If scheduled to use when the instruction will want to use it: Stall

- Stall the conflicting instructions when they try to enter the MEM stage. Heuristic: choose the instruction that came from the unit with the longest latency

  + No need to try to try to figure out the problem at ID time

  - Instructions can now stall before getting to MEM, which complicates the pipeline

How to Handle WAW Hazards?

```
ADDD F2,F4,F6     IF  ID A1 A2  A3  A4   MEM WB
XOR                   IF  ID
LD F2, 0(R2)              IF   ID EX MEM WB
```

- Only if the result of ADDD is not used (compiler generated)
- Strategies:
  1. Delay issue of LD until ADDD enters MEM /*it is known at ID time */
  2. Detect the hazard and prevent ADDD from writing. LD can be issued right away /*again: Known at ID time */

Summary of Multicycle Operations

Do at ID time:

- 1. Check for structural hazard: wait until FU not busy and make sure that register write port will ba available at WB

- 2. Check for RAW hazards: wait until the src regs are not listed as pending destinations

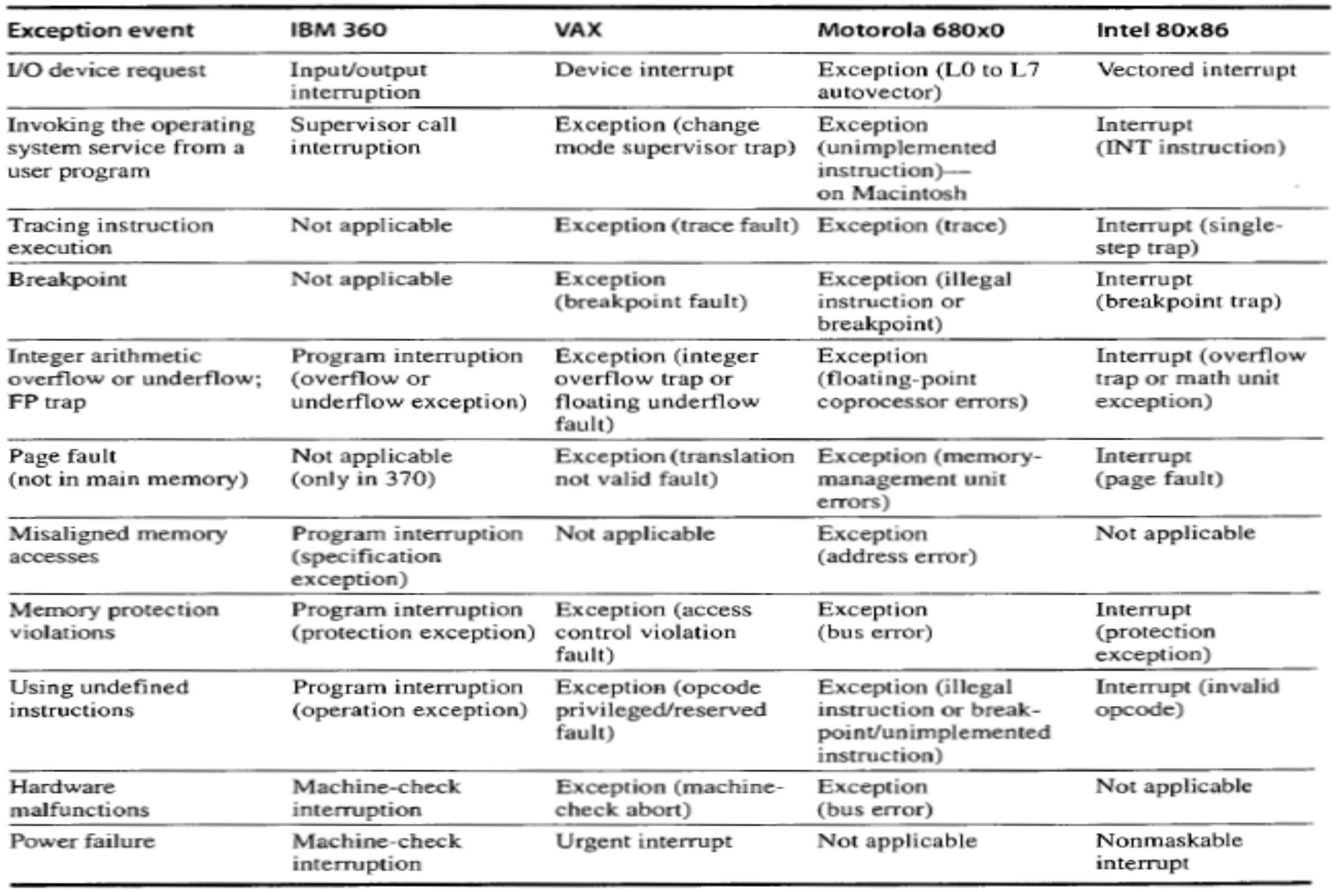
```
IF ID A1 A2 A3 A4 MEM WB
    IF  ID  -    -    -    EX
```

- 3. Check for WAW hazards: Check if any previous instruction has the same register destination as present one. If so, stall until that instr enters the MEM stage

Dealing with exceptions

- Overlapping of instructions in a pipeline: makes it hard to deal with exceptions
- Exceptions:
  - Normal execution order of instructions is changed
  - Include what is sometimes referred to as interrupt and fault
- Examples of exceptions:

  I/O device request, invoking an OS service from a user program, tracing instruction execution, breakpoint, integer arithmetic overflow and underflow, FP arithmetic anomaly, page fault, misaligned memory access, memory protection violation, using an undefined instruction, hardware malfunction, power failure



Exception event	IBM 360	VAX	Motorola 680x0	Intel 80x86
I/O device request	Input/output interruption	Device interrupt	Exception (L0 to L7 autovector)	Vectored interrupt
Invoking the operating system service from a user program	Supervisor call interruption	Exception (change mode supervisor trap)	Exception (unimplemented instruction)—on Macintosh	Interrupt (INT instruction)
Tracing instruction execution	Not applicable	Exception (trace fault)	Exception (trace)	Interrupt (single-step trap)
Breakpoint	Not applicable	Exception (breakpoint fault)	Exception (illegal instruction or breakpoint)	Interrupt (breakpoint trap)
Integer arithmetic overflow or underflow; FP trap	Program interruption (overflow or underflow exception)	Exception (integer overflow trap or floating underflow fault)	Exception (floating-point coprocessor errors)	Interrupt (overflow trap or math unit exception)
Page fault (not in main memory)	Not applicable (only in 370)	Exception (translation not valid fault)	Exception (memory-management unit errors)	Interrupt (page fault)
Misaligned memory accesses	Program interruption (specification exception)	Not applicable	Exception (address error)	Not applicable
Memory protection violations	Program interruption (protection exception)	Exception (access control violation fault)	Exception (bus error)	Interrupt (protection exception)
Using undefined instructions	Program interruption (operation exception)	Exception (opcode privileged/reserved fault)	Exception (illegal instruction or break-point/unimplemented instruction)	Interrupt (invalid opcode)
Hardware malfunctions	Machine-check interruption	Exception (machine-check abort)	Exception (bus error)	Not applicable
Power failure	Machine-check interruption	Urgent interrupt	Not applicable	Nonmaskable interrupt

Figure C.30

Classification

- Synchronous vs asynchronous:
  - Synchronous: event occurs at the same place every time
  - Asynchronous: caused by devices external to processor and memory; can usually be handled after the completion of the current instruction (easier to handle)
- User requested vs user coerced:
  - Requested: user directly asks for it; can always be handled after the instruction has completed
  - Coerced: caused by some hardware event that is not under the control of the user program
- User maskable vs unmaskable:
  - Maskable: a user task can mask it

Classification

- Within vs between instructions: the event prevents instruction completion by happening in the middle of execution or it is recognized between instructions:
  - Within: usually synchronous; harder to implement since instruction must be stopped and restarted; exceptions that are within and asynchronous are usually catastrophic: power failure
- Resume vs terminate
  - Terminate: program terminates. Easier to implement since no need to restart the execution

See Figure C.31

Difficulty

- Implementing interrupts occurring within instructions that are resuming is difficult
- How to do it? Invoke another program that
  - Save the state of the executing program
  - Correct the cause of the exception
  - Restore the state of the program before the I that caused exception
  - Retry the instruction
- Process invisible to program
- If pipeline allows machine to save state, handle exception, and restart without affecting the execution of program: restartable machine or pipeline
- All machines today are restartable (at least for int pipeline)

Stopping and restarting execution

- Focus on exceptions that 1) occur within instruc and 2) are restartable
- Example: page fault from a data fetch
  - Occurs when one instruction enters the MEM stage
  - Other instructions already in the pipeline
  - OS must be invoked, save the state in the pipeline, move page to memory
  - Restore the pipeline as it was and continue
  - If the instruction was a branch: have to reevaluate the condition

Saving the pipeline state safely

- Force a OS instruction into the pipeline on the next IF (the entry point of an OS trap)
- Fill with zeros the latches for the instruction that caused the exception and successors. This prevents any writes and, therefore, changing any state
- Save the PC of the offending instruction, so that we can resume

- Note: suppose we have a 1 branch delay slot and the exception happens in the instruction in the slot? In general, need to save as many PCs as delay slots + 1

Returning machine from exception

- Last instruction in the OS trap handler is a special instruction that returns the PC of the offending instruction
- Therefore, the next instruction to execute is the one causing the exception

- A pipeline supports precise exceptions if: instructions before the faulting one can be completed and those after (and including) it, can be restarted from scratch
  - Easier if the faulting instruction has not updated state (most cases)
  - Harder if it has (e.g. in some FP ops). In this case, the hardware has to retrieve the old values
- Programmers prefer precise interrupts: easy in int pipeline

Exceptions in Integer MIPS

- IF: page fault on I-fetch, misaligned memory access, memory protection violation
- ID: undefined/illegal opcode
- EX: arithmetic exception
- MEM: page fault on D-fetch, misaligned memory access, memory protection violation
- WB: none

Issues

- Two exceptions may occur at the same time:

  LD: in the MEM

  ADD: in the EX

  What to do? Repair one and restart; dectect and repair the second one
- Two exceptions may appear out of order:

  LD: in the MEM

  ADD: in the IF

  What to do? Hardware posts all exceptions caused by an instruction in a status vector assoc with the instruction. The vector is carried down the pipeline with the instr.

Issues (cont)

- Once an exception indication is set in the vector, the instruction cannot update state (regs or mem)
- When instr is about to leave MEM, its vector is checked
- If any bit set, process the exception (if several in the vector, process the one from the earliest stage in the pipeline)

Instruction Set Complications

- Precise exceptions easy in MIPS: no instruction writes until the end of the pipe (end of MEM or WB)
- Other ISAs have instructions that change the state in the middle:
  - Autoincrement updates a reg in the middle, and then it can suffer an exception
    - In that case, need extra hardware support to undo the change, to reset the state to before the instruction (or the exception will be imprecise)
    - Alternatively, do not let any update until the instruction commits (it is guaranteed to complete), but it may be complex
  - Instructions that update memory during execution (string copy)
    - State of partially completed instruction always in registers

Exceptions with Long-Running Instr

```
DIVF    F0,F2,F4
ADDF  F10,F10,F8
SUBF   F12,F12, F14
```

- Out of order completion: ADDF, SUBF finish before DIVF
- Suppose that ADDF completes and then DIVF has an EX exception
- Used possible solutions:
  - 1. Ignore the problem and use imprecise interrupts. Need to re-execute the code in a much slower “precise exception” mode. This mode does not allow as much overlap → only 1 FP active at a time

Possible solutions (cont)

- 2. Buffer the result of the FP op until all the FP operations that were issued before are complete
  - Expensive buffering of much state
  - Buffered results may need to be read by subsequent instr
  - Used in the form of history buffer and future file
- 3. Keep enough information around so that trap-handling routines can create a precise interrupt: software finishes all the instructions that precede the one that finished (ADDF)
  - Quite complicated
  - Simplified if we have only 2 FPs executing at a time

Possible solutions (Cont)

- 4. Allow instruction issue (entering the EX stage) only when we know that all the previous instr issued will finish without exceptions
  - Guarantees precise interrupts
  - It implies often stalling before issuing
  - For it to work: the FP pipeline must decide soon (in 2-3 EX cycles) if there will be an exception