UT 2. Pipelined Computers

Lecture 2.1 Pipelined instruction units.

J. Duato, J. Flich, P. López, V. Lorente,
A. Pérez, S. Petit, J.C. Ruiz, S. Sáez, J. Sahuquillo

Department of Computer Engineering
Universitat Politècnica de València

Contents

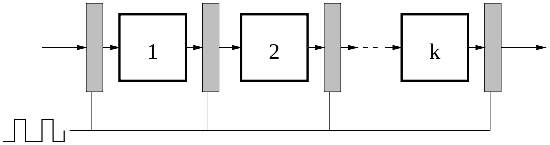
Bibliography

John L. Hennessy and David A. Patterson.
Computer Architecture, Fifth Edition: A Quantitative Approach.
Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5 edition, 2012.

Contents

Pipelining (1/3)

- a process is decomposed into several subprocesses
- each subprocess is independently executed in an autonomous module
- each module works concurrently with the rest



$\rightarrow$ Stage: module processing a subtask (with a certain delay)

+

latch register

Pipelining (2/3)

- **Latches**:
  They keep data stable during the time required by a module to perform its function.
- A clock synchronizes the advance of data among stages. The clock defines
  - when new data can enter the pipelined unit.
  - the time available for each stage to perform its function.

Pipelining (3/3)

- **Clock period**
  - Ideal case: same delay in all the modules

$$\tau = \frac{D}{k}$$

    - ★ D: original circuit delay
    - ★ k: number of stages
  - Real case: modules with variable delays, latches and clock skew.

$$\tau = \max_{i=1}^{k}(\tau_i) + T_R + T_S \geq \frac{D}{k}$$

    - ★ τ_i: Module delay i.
    - ★ T_R: Inter-stage latch (register) delay.
    - ★ T_S: Clock skew.

Benefits of pipelining (1/2)

- ***Speed-up***:

$$S = \frac{T_{np}}{T_p}, \text{given:}$$

  - T_{np}: Time to process n instructions in the original unit.
  - T_p: Idem in the pipelined unit.
  - Ideal case: $\tau = \frac{D}{k}$.

$$S = \frac{D}{\tau} = k$$

  - Real case: $\tau \geq \frac{D}{k}$.

$$S = \frac{D}{\tau} \leq k$$

Benefits of pipelining (2/2)

- **Throughput**
  - General expression:

$$\chi = \frac{n}{T}, \text{where:}$$

    - ⋆ n: processed data.
    - ⋆ T: required processing time for n data.
  - Non-pipelined unit.

$$\chi_{ns} = \frac{n}{T_{ns}} = \frac{n}{nD} = \frac{1}{D} \text{ results/s}$$

  - Pipelined unit:

$$\chi_{s} = \frac{n}{T_{s}} \approx \frac{n}{n\tau} = \frac{1}{\tau} \text{ results/s}$$

$\rightarrow$ 1 result each τ seconds = 1 results per clock cycle.

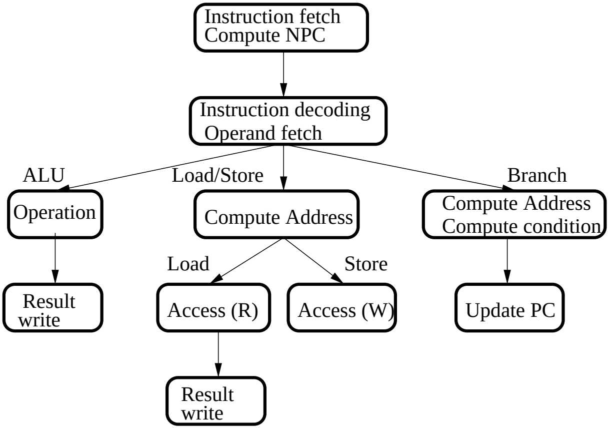
Contents

Example computer: Simplified MIPS

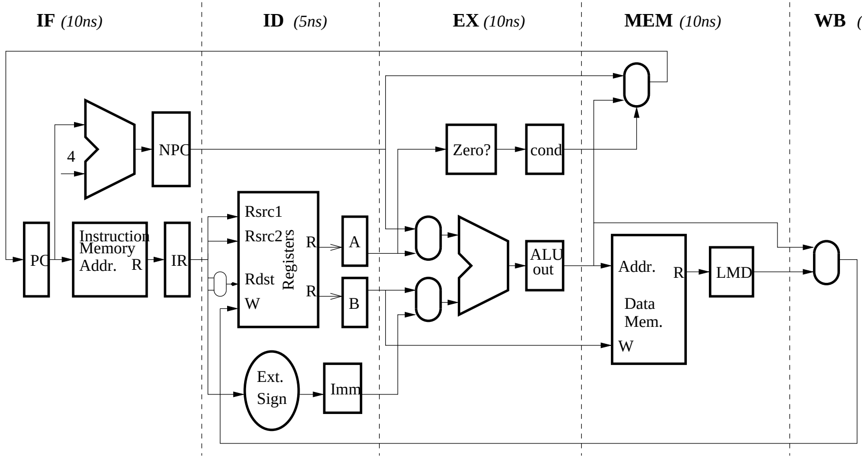
- Arithmetic instructions (reg–reg and reg–imm).
- Loads and stores.
- Conditional branches `beqz rs,desp(PC)` and `bnez rs,desp(PC)` (if rs= or $\neq$ 0 and offset of 16 bits).

	0 5	6 10	11 15	16 20	21 31
R	Op. Code	R_{src1}	R_{src2}	R_{dst}	Func.
	$\Rightarrow$ ALU (reg–reg) instr.: $R_{dst} \leftarrow R_{src1}$ op R_{src2}				
	0 5	6 10	11 15	16 31	
I	Op. Code	R_{src1}	R_{dst}	Imm	
	$\Rightarrow$ ALU (reg–imm) instr.: $R_{dst} \leftarrow R_{src1}$ op Imm $\Rightarrow$ Load: $R_{dst} \leftarrow$ M[R_{src1}+Imm] $\Rightarrow$ Store: M[R_{src1}+Imm] $\leftarrow R_{dst}$ $\Rightarrow$ Conditional branch: if (R_{src1} =,$\neq$ 0) then PC $\leftarrow$ PC+4+Imm				

MIPS instruction cycle



MIPS datapath



Contents

MIPS pipeline

Problem

Stages executed by instructions varies from one instructions to another.

Solution: assume that all instructions traverse the same stages.

- IF, ID, EX, MEM, WB
- some instructions *do nothing* in some stages.

Execution of instructions

i	IF	ID	EX	MEM	WB				
$i+1$		IF	ID	EX	MEM	WB			
$i+2$			IF	ID	EX	MEM	WB		
$i+3$				IF	ID	EX	MEM	WB	
$i+4$					IF	ID	EX	MEM	WB

Speed-up:

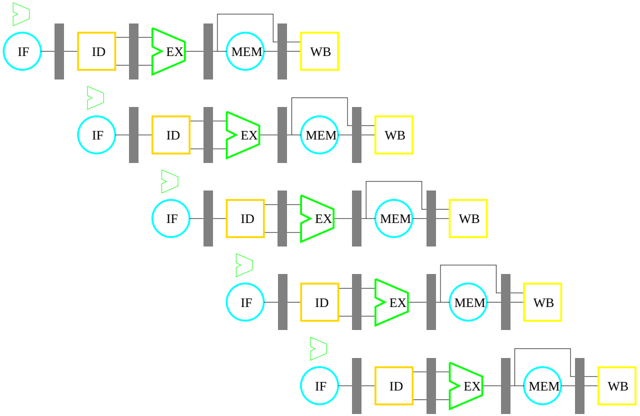
	Non-pipelined	Pilelined
I	n	n
CPI	1	1
T	$\max(30, 40) = 40$ ns	$\max(5, 10) = 10$ ns
$T_{\text{ej}} = I \cdot CPI \cdot T$	$n \cdot 1 \cdot 40 = 40n$ ns	$n \cdot 1 \cdot 10 = 10n$ ns

$$S = 4$$

The number of stages $k = 5$ is the speed-up upper-bound

Hardware requirements: (1/2)

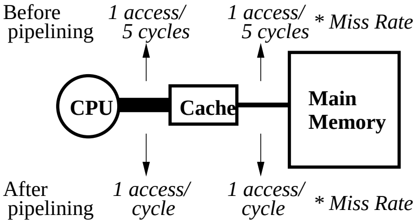
- In a single cycle there are 5 instructions simultaneously in execution → avoid hazards in functional units.



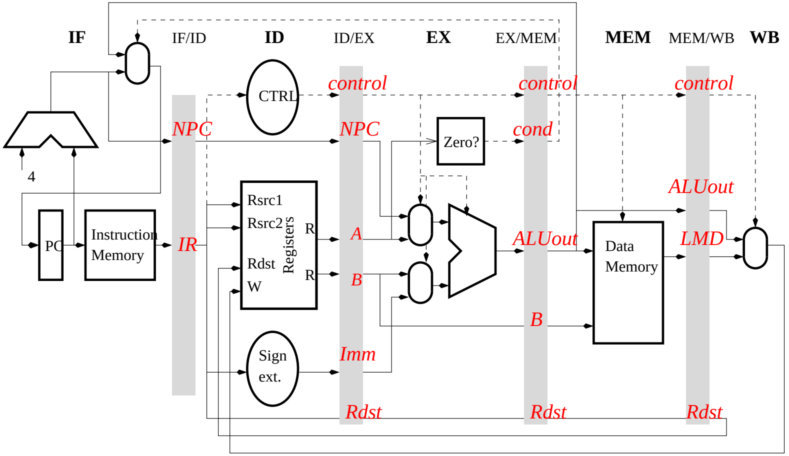
- One operator (EX) and one adder (IF).

Hardware requirements: (2/2)

- Separated instruction (IF) and data (MEM) caches.
- Register file with two simultaneous read (ID) and one write (WB) ports.
- *Cache* access time remains constant, but the required bandwidth is 5 times larger:

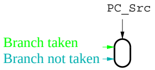
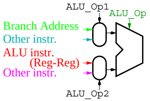
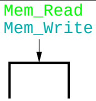
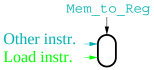


Pipelined datapath



Contents

Control signals (1/2)

IF	ID	EX	MEM	WB
Mem_Read PC_Src	Reg_Read	ALU_Op ALU_Op1 ALU_Op2	Mem_Read Mem_Write	Reg_Write Mem_to_Reg
				

Control signals (2/2)

- Required control signals in EX, MEM and WB stages are generated at stage ID and they travel through the pipelined unit.
- Identifier of destination register follows the same path as the data to be written into. Both informations are simultaneously provided to the register file.
- PC management logic is moved to stage IF (PC is incremented every cycle)

Events and signals in the pipelined instruction cycle (1/3)

	All instructions

IF	IF/ID.IR $\leftarrow$ Mem[PC]
	IF/ID.NPC $\leftarrow$ PC+4
	if EX/MEM.cond then PC $\leftarrow$ EX/MEM.ALUout
	else PC $\leftarrow$ PC+4

ID	ID/EX.A $\leftarrow$ Regs[IF/ID.IR$_{6..10}$] ID/EX.B $\leftarrow$ Regs[IF/ID.IR$_{11..15}$] ID/EX.Imm $\leftarrow$ ((IF/ID.IR$_{16}$)^{16} ## IF/ID.IR$_{16..31}$ ID/EX.NPC $\leftarrow$ IF/ID.NPC ID/EX.IR $\leftarrow$ IF/ID.IR

Events and signals in the pipelined instruction cycle (2/3)

	ALU Instr. Reg–Reg/Reg–Imm
EX	EX/MEM.IR ← ID/EX.IR EX/MEM.ALUout ← ID/EX.A op ID/EX.B || EX/MEM.ALUout ← ID/EX.A op ID/EX.Imm EX/MEM.cond ← 0
MEM	MEM/WB.IR ← EX/MEM.IR MEM/WB.ALUout ← EX/MEM.ALUout
WB	Regs[MEM/WB.IR$_{16..20}$] ← MEM/WB.ALUout || Regs[MEM/WB.IR$_{11..15}$] ← MEM/WB.ALUout

Events and signals in the pipelined instruction cycle (3/3)

	Load/*Store*
EX	EX/MEM.IR ← ID/EX.IR EX/MEM.ALUout ← ID/EX.A + ID/EX.Imm EX/MEM.cond ← 0 EX/MEM.B ← ID/EX.B
MEM	MEM/WB.IR ← EX/MEM.IR MEM/WB.LMD ← Mem[EX/MEM.ALUout] ‖ Mem[EX/MEM.ALUout] ← EX/MEM.B
WB	Regs[MEM/WB.IR$_{11..15}$] ← MEM/WB.LMD ‖

	Branch BEQZ/*BNEZ*
EX	EX/MEM.ALUout ← ID/EX.NPC + ID/EX.Imm EX/MEM.cond ← ID/EX.A op 0 ‖ $\neq$ 0

Contents

Concept and classification

Hazard : Situation leading to the execution of some instructions generating results that are not consistent with the ones produced by the non-pipelined datapath $\Rightarrow$ Lost of binary compatibility.
Hazards originated by two or more instructions simultaneously present in the pipelined instruction unit.

Hazard types :

Data	$\rightarrow$ The result of one instruction is used as input data in the following one(s).
Control	$\rightarrow$ Branch instructions modify the flow of instructions
Structural	$\rightarrow$ Two or more instructions want to use the same resource

Hazard solutions (1/2)

Stall insertion Stop the instruction originating a hazard, and the following ones, but continue with the execution of those preceding such instructions (otherwise the hazard will never disappear)

- During these cycles no instruction is fetched (*stalls*).
- Performance degradation:

$$\mathrm{CPI}_s = \mathrm{CPI}_{s_{\mathrm{ideal}}} + \frac{\text{stalls}}{\text{instruction}} = 1 + \frac{\text{stalls}}{\text{instruction}} > 1$$

Hazard solutions (2/2)

Datapath modification
: Modify the *hardware* to dynamically solve hazards

- Reduce the number of stalls required to solve the problem.
- Complete solution of the problem (it is sometimes impossible).

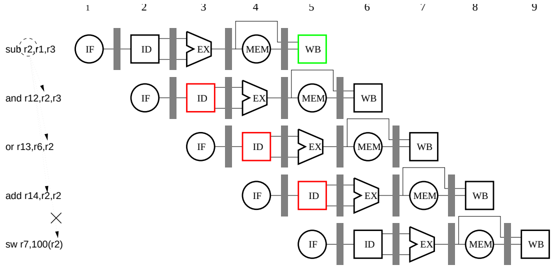
Compiler modification
: Avoid the problem by avoiding the generation of certain sequences of instructions

- Drawback: Binary compatibility with datapaths that accept any sequence of instructions is lost.

Contents

Causes

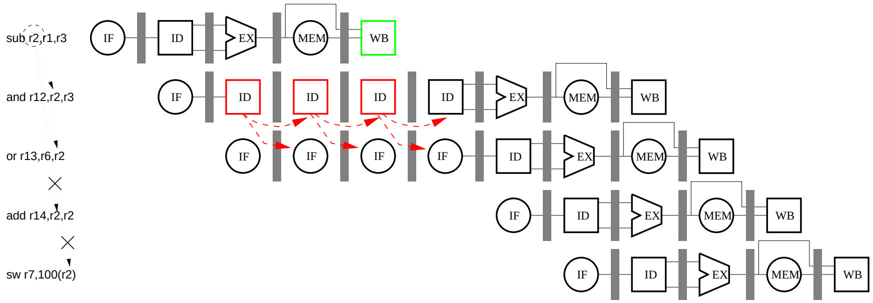
Some instructions rely on the results of previous ones → pipelining the instruction unit may change operand access order



Hazards can involve up to 4 consecutive instructions

Stall insertion

Delay those operations originating the hazard



→ 3 *stalls* for every two dependent and consecutive instructions.
→ 2 *stalls* for every two dependent instr. separated by 1 instruction.
→ 1 *stall* for every two dependent instr. separated by 2 instructions.
→ significant loss of performance

Reduction of the number of instructions involved in the hazard (1/2)

$\Rightarrow$ Datapath modification

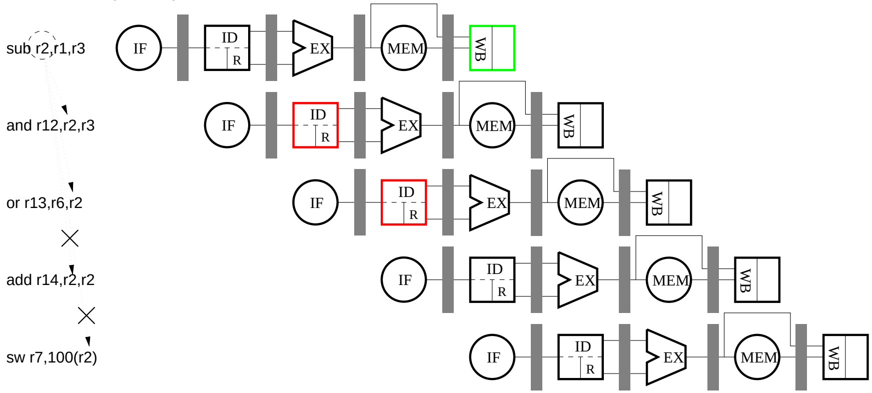
- Multiport register file supporting simultaneous read and write
- Modification of the access time to the register file:

  Register read $\rightarrow$ Second part of the cycle

  Register write $\rightarrow$ First part of the cycle.

  - There is no problem if the register file access time is $\leq$ half of the clock period.
  - Simplifies the design of the register file: Simultaneous read and write operations are not required.

Reduction of the number of instructions involved in the hazard (2/2)



⇒ The number of involved instructions is 3: **dsub**, **and** y **or**.

→ Penalty generated by dependent consecutive instructions: 2 *stalls*

Insertion of 2 stalls (1/3)

```
dsub r2,r1,r3     IF ID EX ME WB
and r12,r2,r3        IF id id ID EX ME WB
or r13,r6,r2            if if IF ID EX ME WB
dadd r14,r2,r2                   IF ID EX ME WB
sd 100(r2),r7                       IF ID EX ME WB
```

- Detect those situation (in bold) requiring stall insertion.
- In order to insert a stall:
  - Set control signals from ID to EX as they were the ones of a `NOP` instruction.
  - Preserve instruction in IF and ID.

Insertion of 2 stalls (2/3)

Control Logic

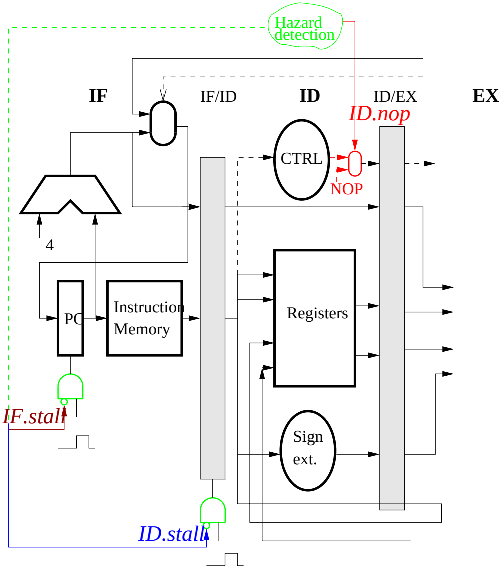
- First cycle:

```
if (($ID/EX.IR_{CODOP}$ = "ALU") and
    ($IF/ID.IR_{CODOP}$ = "ALU") and
    (($ID/EX.IR_{Rdst}$ = $IF/ID.IR_{src1}$) or ($ID/EX.IR_{Rdst}$ = $IF/ID.IR_{Rsrc2}$)
then
    IF.stall, ID.stall, ID.nop
```

- Second cycle:

```
if (($EX/ME.IR_{CODOP}$ = "ALU") and
    ($IF/ID.IR_{CODOP}$ = "ALU") and
    (($EX/ME.IR_{Rdst}$ = $IF/ID.IR_{Rsrc1}$) or ($EX/ME.IR_{Rdst}$ = $IF/ID.IR_{Rsrc2}$
then
    IF.stall, ID.stall, ID.nop
```

Insertion of 2 stalls (3/3)



Forwarding (1/6)

Problem: 2 *stalls* between consecutive instructions due to a data hazard

```
dsub r2,r1,r3    IF ID EX ME WB
and r12,r2,r3       IF id id ID EX ME WB
or r13,r6,r2           if if IF ID EX ME WB
```

Consider the hazard between the 2 first instructions if no stalls were inserted:

```
                  1  2  3  4  5
dsub r2,r1,r3    IF ID EX ME WB
and r12,r2,r3       IF ID EX ME WB
```

- `and` requires the data at the beginning of cycle 4 (its EX stage)
- `dsub` provides a result at the beginning of cycle 4 (its MEM stage)

Forwarding (2/6)

⇒ datapath modification

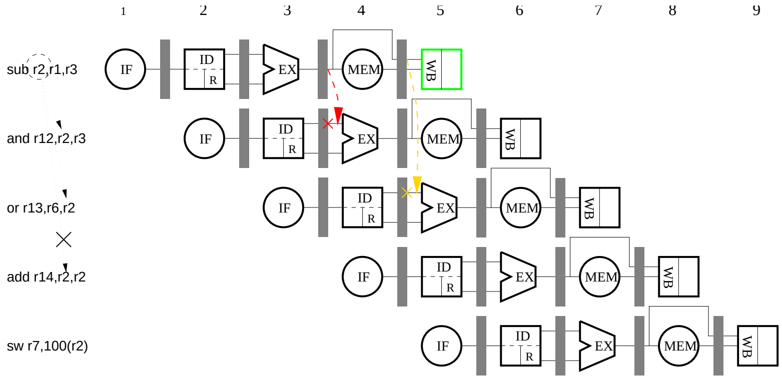
Add a bus from the ALU output (MEM stage) to its inputs (EX stage) + control logic ⇒ “short-circuit” from MEM to EX.

When the third instruciton is considered, the hazard can be solved in a similar way, i.e. implementing a short-circuit between WB and EX.

```
dsub r2,r1,r3    IF ID EX ME WB
and r12,r2,r3       IF ID EX ME WB
or r13,r6,r2           IF ID EX ME WB
```

Forwarding (3/6)

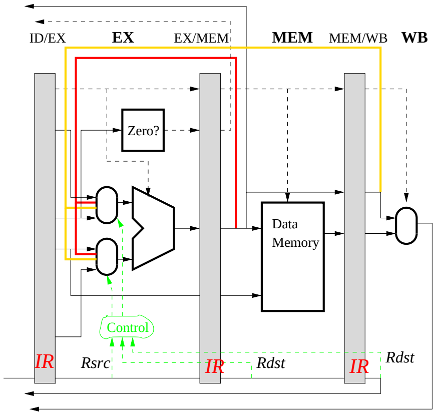
Following instructions requiring the data will not need any shortcircuit.



Forwarding (4/6)

Forwarding implementation

- Multiplexors in the ALU input



Forwarding (5/6)

Control logic

- Shortcircuit from MEM to EX:
  - Rsrc1:

```
if ((EX/MEM.IR$_{OPCODE}$ = "ALU") and
    (ID/EX.IR$_{OPCODE}$ = "ALU") and
    (EX/MEM.IR$_{Rdst}$ = ID/EX.IR$_{Rsrc1}$))
then
    Shortcircuit MEM-to-EX (high part of ALU inputs)
```

  - Rsrc2:

```
if ((EX/MEM.IR$_{OPCODE}$ = "ALU") and
    (ID/EX.IR$_{OPCODE}$ = "ALU") and
    (EX/MEM.IR$_{Rdst}$ = ID/EX.IR$_{Rsrc2}$))
then
    Shortcircuit MEM-to-EX (low part of ALU inputs)
```

Forwarding (6/6)

- Shorcircuit from WB to EX:
  - Rscr1:

```
if ((MEM/WB.IR$_{OPCODE}$ = "ALU") and
    (ID/EX.IR$_{OPCODE}$ = "ALU") and
    (MEM/WB.IR$_{Rdst}$ = ID/EX.IR$_{Rsrc1}$))
then
   Shorcircuit WB-to-EX (high part of ALU inputs)
```

  - Rsrc2:

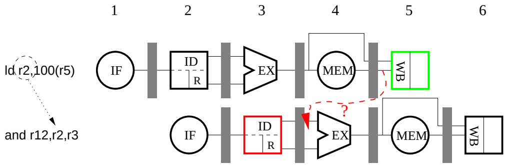
```
if ((MEM/WB.IR$_{OPCODE}$ = "ALU") and
    (ID/EX.IR$_{OPCODE}$ = "ALU") and
    (MEM/WB.IR$_{Rdst}$ = ID/EX.IR$_{Rsrc2}$))
then
    Shorcircuit WB-to-EX (low part of ALU inputs)
```

Load-dependent and consecutive instruction (1/3)

Consider the hazard between the following instructions:

ld r2,100(r5)
and r12,r2,r5

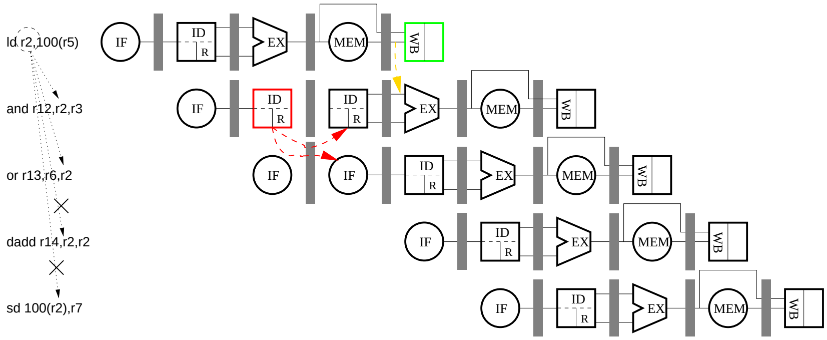
- When does **and** require the data? In cycle 4 (EX stage)
- When is the result of **ld** available? In cycle 5 (WB stage)



⇒ Cannot be solved using only data forwarding

Load-dependent and consecutive instruction (2/3)

Even using a shortcircuit from WB to EX , it is necessary to insert 1 stall



Load-dependent and consecutive instruction (3/3)

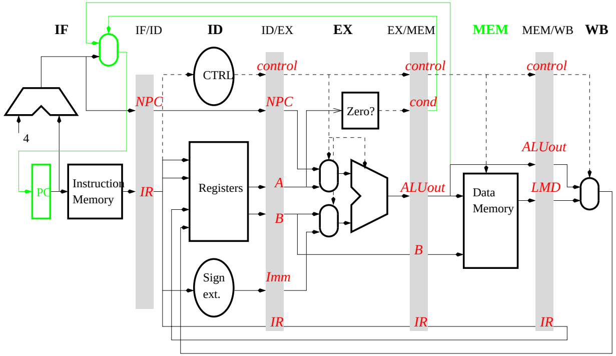
Required control logic to insert the stall

```
if ((ID/EX.IR$_{OPCODE}$ = "LOAD") and
    (IF/ID.IR$_{OPCODE}$ = "ALU") and
    (ID/EX.IR$_{Rdst}$ = IF/ID.IR$_{Rsrc}$))
then
    ID.stall, IF.stall, ID.nop
```

Contents

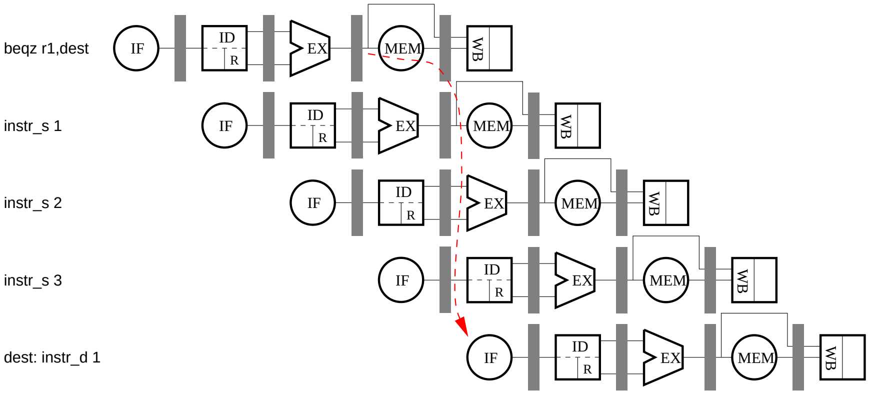
Causes (1/2)

In the instruction flow, some instructions modify the value of the PC.
Branch instructions modify the PC in the MEM stage.



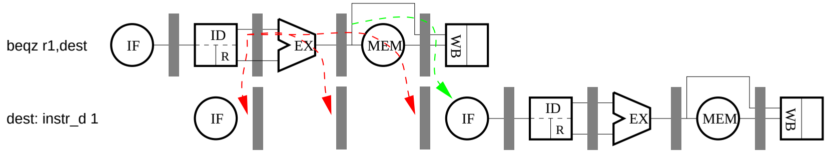
Causes (2/2)

When this cycle is reached, 3 instructions have already been issued:



Stall insertion (1/2)

Insert stalls *whenever* a branch instruction is decoded:



3 stalls $\rightarrow$ loss of performance

Stall insertion (2/2)

Control logic for the 3 stalls to insert

- Disable IF stage during 3 cycles, thus sending `NOP` to ID:

```
beqz r1,dest    IF ID EX ME WB
<dest>             if if if IF ID EX ME WB
```

```
if ((IF/ID.IR_OPCODE = "Branch") or
    (ID/EX.IR_OPCODE = "Branch") or
    (EX/MEM.IR_OPCODE = "Branch"))
then
    IF.stall, IF.nop
```

(Fixed) prediction (1/2)

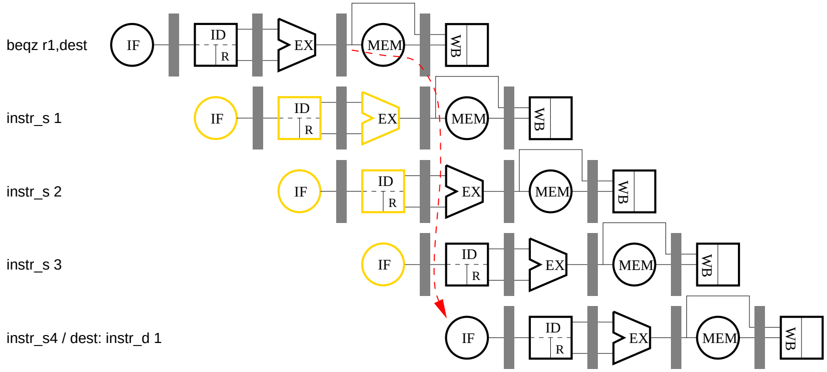
⇒ Datapath modification

Predict-not taken Assume the branch is not taken → Instructions following the branch are correct.
At the end, if the branch is taken, these 3 instructions must be cancelled.
IMPORTANT: These instructions must not modify the computer state

Predict-taken Assume the branch is taken → once the destination address is known, new instructions are fetched
If the branch is finally not taken, such instructions are aborted.
It is only useful if the destination address is known *before* the branch condition → not useful in the case of the MIPS.

(Fixed) prediction (2/2)

Predict-not-taken



Control logic

```
if (EX/MEM.cond) then
    IF.nop, ID.nop, EX.nop
```

Reducing the branch latency (1/5)

⇒ Datapath modification

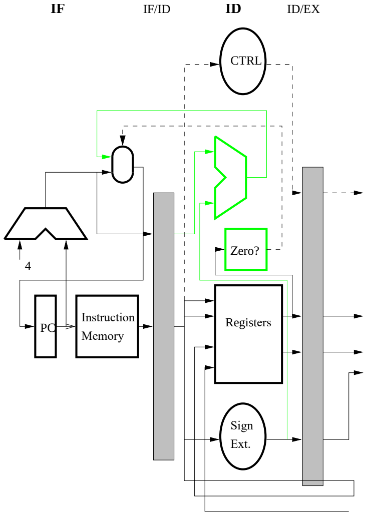
Reduce the number of cycles between fetching the branch instruction and computing the branch destination.

- Computation of destination address moves from EX to ID → need of an additional adder
- Evaluation of the condition moves from EX to ID.
- Update the PC in ...
    - ... stage EX → num. of cycles = 2
    - ... stage ID (at its end) → num. of cycles = 1

      ⇒ In stage ID the new PC is known

Reducing the branch latency (2/5)

Updating the PC in the ID stage



Reducing the branch latency (3/5)

Control logic for updating the PC in the ID stage

- Assuming *predict-not-taken*, if the branch is not taken, fetch the next instruction.
- But if the branch is taken, fetched instruction must be cancelled during IF. → A NOP is provided to the ID stage.

```
beqz r1,dest    IF **ID** EX ME WB
<sgte>             **IF** X
<dest>                IF ID EX ME WB
```

```
if (IF/ID.IR_OPCODE = "Branch")
then
    if (Regs[IF/ID.IR_Rsrc] op 0)
    then
        IF.nop
        PC <- IF/ID.NPC + IF/ID.Imm
    else
        PC <- PC + 4
```

Reducing the branch latency (4/5)

Impact on the clock period

Stage ID

register access time +
delay for evaluating the condition +
selection +
PC update
$\rightarrow$ ID may become the slowest stage
$\Rightarrow$ branch condition must be simple ($=$ and $\neq$).

This modification may be incompatible with the requirement of reading the registers in half a cycle.

Reducing the branch latency (5/5)

Impact on stalls produced by data hazards

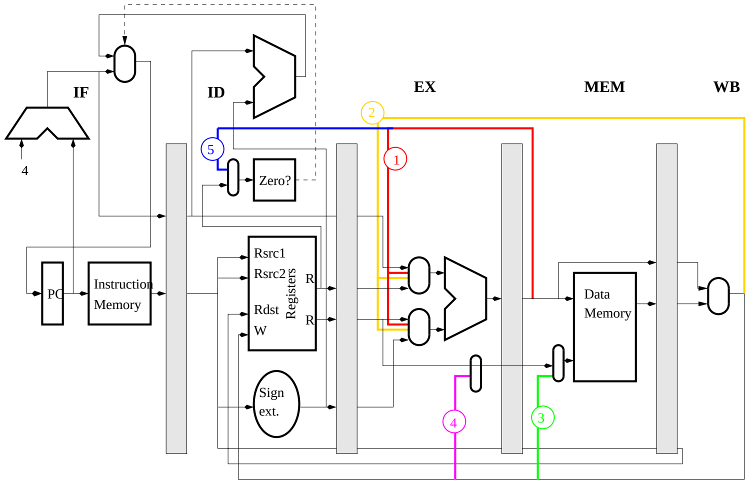
Data hazards where branches are involved require he use of short-circuits in the ID stage and/or the insertion of stalls. Examples:

```
dslt r1,r2,r3   IF ID EX ME WB
beqz r1,loop       IF id ID EX ME WB

dslt r1,r2,r3   IF ID EX ME WB
...
beqz r1,loop          IF ID EX ME WB

ld r1,20(r2)    IF ID EX ME WB
beqz r1,loop       IF id id ID EX ME WB

ld r1,20(r2)    IF ID EX ME WB
...                IF ID EX ME WB
beqz r1,loop          IF id ID EX ME WB
```

Possible data hazards

Instrucciones	Ejemplo	Cortocircuito	*stalls*	Fig.
ALU - ALU	DADD R1,R2,R3			
	DSUB R4,R1,R5	MEM to EX	0	1
	AND R7,R1,R6	WB to EX	0	2
Load - ALU	LD R1,20(R2)			
	DADD R3,R1,R4	WB to EX	1	2
ALU - Load/Store	DADD R1,R2,R3			
	LD R2,20(R1)	MEM to EX	0	1
	LD R3,40(R1)	WB to EX	0	2
ALU - Store	DADD R1,R2,R3			
	SD R1,20(R2)	WB to MEM	0	3
	SD R1,40(R2)	WB to EX	0	4
Load - Store	LD R1,20(R3)			
	SD R1,20(R2)	WB to MEM	0	3
	SD R1,40(R2)	WB to EX	0	4
Load - Load/Store	LD R1,30(R3)			
	LD R2,20(R1)	WB to EX	1	2
ALU - Branch	DSLT R1,R2,R3			
	BEQZ R1,loop	MEM to ID	1	5
ALU - Branch	DSLT R1,R2,R3			
	...			
	BEQZ R1,loop	MEM to ID	0	5
Load - Branch	LD R1,20(R2)			
	BEQZ R1,loop	–	2	–
Load - Branch	LD R1,20(R2)			
	...			
	BEQZ R1,loop	–	1	–

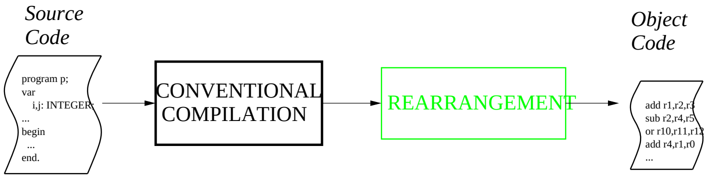
Modified datapath



Contents

Concept (1/2)

⇒ Addition of a new step to the compilation process: code rearrangement



- Enables getting by without circuits specifically designed for solving hazards.
  → Possible reduction of the *CPI* and *t* at the cost of binary compatibility loss.
- Even if those circuits exist, code rearrangement can prevent stalls.
  → *CPI* reduction.

Concept (2/2)

- If code rearrangement is not possible, NOP instructions are inserted.
  $\rightarrow$ slightly increase of *I*.
  $\rightarrow$ In the worst of the case, similar performance to the insertion of stalls.

Two examples:

- For data hazards $\rightarrow$ *Delayed load*.
- For control hazards $\rightarrow$ *Delayed branch*.

Delayed load (1/2)

Even when using shortcircuits, an instruction just following a load and presenting a data hazard with such load always requires a stall. Example:

```
ld r2,100(r5)    IF ID EX ME WB
and r12,r2,r5       IF ID ID EX ME WB
```

Delayed load: The compiler ensures that no instruction following a load reads from the loaded register.

Load delay slot

- Number of instructions after a load that cannot read from the loaded register.
  $\rightarrow$ This number depends on the existing instruction pipeline.
  $\rightarrow$ 1 cycle in the case of the MIPS.

Delayed load (2/2)

Example

Conventional code	Code with `NOP`s	Rearranged code
LD Rb,b	LD Rb,b	LD Rb,b
LD Rc,c	LD Rc,c	LD Rc,c
DADD Ra,Rb,Rc	NOP	LD Re,e
LD Re,e	DADD Ra,Rb,Rc	DADD Ra,Rb,Rc
LD Rf,f	LD Re,e	LD Rf,f
DSUB Rd,Re,Rf	LD Rf,f	SD Ra,a
SD Ra,a	NOP	DSUB Rd,Re,Rf
SD Rd,d	DSUB Rd,Re,Rf	SD Rd,d
	SD Ra,a	
	SD Rd,d	
8 ins + 2 stalls = 10 cycles	10 ins = 10 cycles	8 ins = 8 cycles

Delayed branch (1/5)

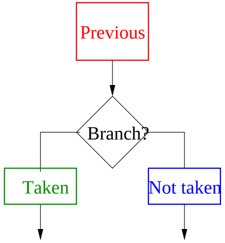
The compiler places instructions after branches that will be executed regardless of whether branches are taken or not.

- Since these instructions will be always executed, they do not require any cancellation or the insetion of any stall.
- In general, the compiler selects instructions *preceding* the branch that do not present any hazard with the branch condition.

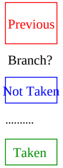
Branch delay slot

- Number of instructions after the branch that will be always executed despite the branch will be taken or not.
  $\rightarrow$ This value is equal to the branch latency.
  $\rightarrow$ 1 cycle if the PC is updated during the ID stage.

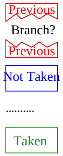
Delayed branch (2/5)



Conventional code:



Code with delayed branch:



Example:

Conventional	Delayed branch
`ADD Rc,Ra,Rb`	`BEQZ Ra,dst`
`BEQZ Ra,dst`	`ADD Rc,Ra,Rb`
`INSTR1`	`INSTR1`
`dst: INSTR2`	`dst: INSTR2`

Delayed branch (3/5)

- The use of instructions presenting hazards with the branch condition is acceptable if such instructions have a high-probability of being executed. Example:

```
                                     LD    Ra,A(Ri)
cont: LD    Ra,A(Ri)           cont: DADDI Ri,Ri,-1
      DADDI Ri,Ri,-1      →          DADD  Rb,Rb,Ra
      DADD  Rb,Rb,Ra                 BNEZ  Ri,cont
      BNEZ  Ri,cont                  LD    Ra,A(Ri)
```

- What if it is not possible to find out such type of instructions? Use of `NOPs` instead.

Delayed branch (4/5)

Compilation strategies

Where are the instruction(s) used to fill the *delay slot* coming from?

Instruction	Conventional	Delayed Branch
Preceding	ADD R1,R2,R3	...
	BEQZ R2,dst	BEQZ R2,dst
	...	ADD R1,R2,R3
Following	ADD R2,R2,R3	ADD R2,R2,R3
	BEQZ R2,dst	BEQZ R2,dst
	...	OR R7,R8,R9
	OR R7,R8,R9	. . .
	...	...
	dst: SUB R4,R5,R6	dst: SUB R4,R5,R6
At destination	ADD R2,R2,R3	ADD R2,R2,R3
	BEQZ R2,dst	BEQZ R2,dst
	...	SUB R4,R5,R6
	...	...
	OR R7,R8,R9	OR R7,R8,R9
	...	...
	dst: SUB R4,R5,R6	dst: AND R10,R11,R12
	AND R10,R11,R12	...

Delayed branch (5/5)

Strategy	Restrictions	Performance improvement
Preceding	Branch condition does not depend on the instruction.	Always
Following	If condition is met, it must not lead to any problem even when executed.	If the condition is not met.
At destination	If condition is not met, it must not lead to any problem even when executed. If the destination can be reached from other points, it is also necessary to copy the instruction to them.	If the condition is met.

Contents

Causes

- The hardware does not allow all possible combinations between instructions in the unit.
  $\rightarrow$ A resource has not been replicated enough
- **Example**: Processor with single instruction–data cache.
  $\rightarrow$ The MEM stage of load/store instructions may collide with the IF stage of instructions fetched 3 cycles later

Solutions

Datapath modifications

Replicate the resource in order to enable such combination of instructions.
$\rightarrow$ Example: *Harvard* architecture: it uses separated instruction and data caches.
$\rightarrow$ Cost increase
$\rightarrow$ Replicating the resource is not always possible or makes sense

Stall insertion

Delay the instructions generating the hazard
$\rightarrow$ *stalls* $\rightarrow$ Loss of performance
$\Rightarrow$ The best approach depends on the % of each type of structural hazard

Stall insertion (1/3)

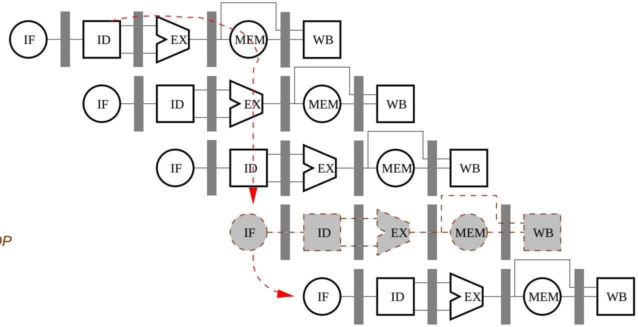
ld r1,100(r0)

i+1

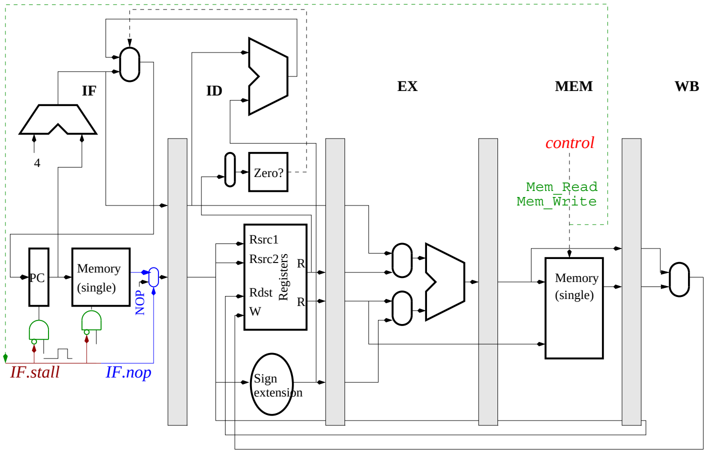
i+2

i+3 → *NOP*

i+3



Stall insertion (2/3)



Stall insertion (3/3)

Control logic

When a load/store instruction is in MEM

- No instruction fetch is performed
- The instruction in IF is stalled
- A `NOP` is deliverd to stage ID

```
if ((EX/MEM.Mem_Read) or (EX/MEM.Mem_Write))
then
  IF.stall,  IF.nop
```

Contents

Concept and classification

Names: Interrupt, **exception.**
Types:

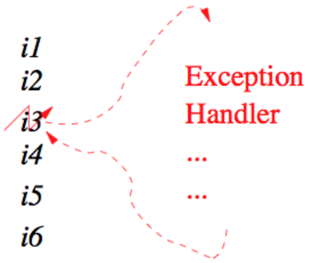
- Synchronous *vs.* asynchronous. It is synchronous if the event is triggered at the same location on every program execution.
- User-driven *vs.* raised to the user.
- Maskable *vs.* unmaskable.
- During the execution of an instruction *vs.* between instructions.
- Continue the program execution *vs.* end the program.

Most difficult ones: Exceptions raised to the user that are provoked during the execution of instructions, when the program continues its execution.

MIPS possible exceptions

Stage	*Exceptions*
IF	Instruction page fault, Misaligned access Violation of protection, E/S request
ID	Ilegal instruction, E/S request
EX	Arithmetic exception, E/S request
MEM	Data page fault, Misaligned access Violation of protection, E/S request
WB	E/S request

Exceptions in conventional computers

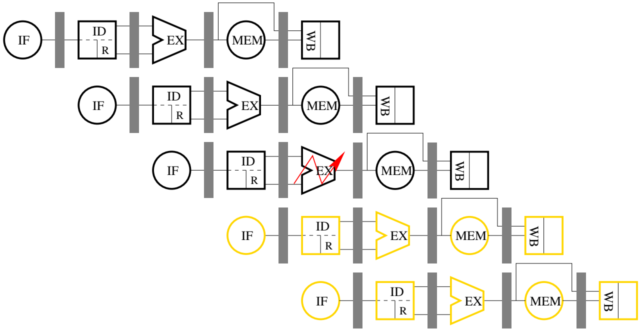


Correct sequence: ... $i1$, $i2$, $i3$ - handler - $i3$, $i4$, $i5$, $i6$ ...

Exceptions in pipelined computers (1/2)

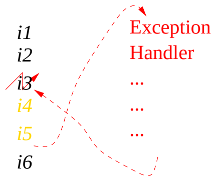
Several instructions under execution when the exception occurs.
Problem:

- There are instructions following the one raising the exception.



Exceptions in pipelined computers (2/2)

Behaviour is incorrect:



Sequence: . . . *i*1, *i*2, *i*3, *i*4, *i*5 - handler - *i*3, *i*4, *i*5, *i*6 . . .

- The PC of instructions is only kept until ID.

First pipelined machines terminated the execution of programs by printing the PC of the current instruction in IF
$\Rightarrow$ they signaled *aproximately* the instruction originating the exception
$\rightarrow$ *imprecise* exceptions.

Precise exceptions in pipelined instruction units

A computer supports a *precise* behaviour in the presence of an exception if:

- Instructions preceding the one generating the exception correctly finish their execution.
- The instruction raising the exception and the following ones are aborted.
- After handler completion it is possible to restart the program from the instruction originating the exception.

$\Rightarrow$ It is possible to identify the instruction raising the exception.
$\Rightarrow$ The behavior is identical to the one exhibited by a non-pipelined computer.

Implementation of precise exceptions in the MIPS (1/4)

Requirements:

- More than one exception (up to 5) can be raised
- ... in the same or in different clock cycles.
- It is necessary to take into account how the branch delay technique works.

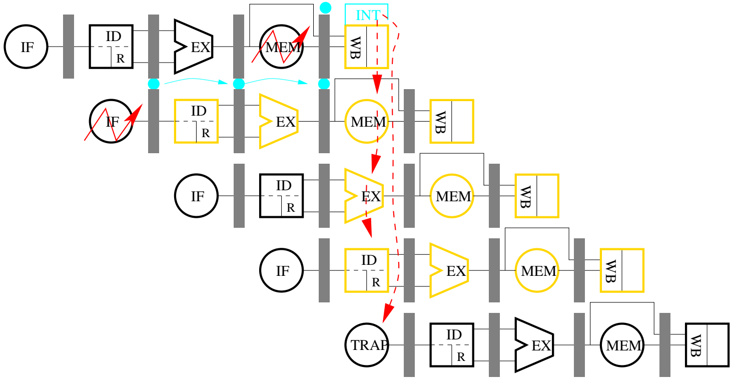
Idea: Ensure a natural order when handling exceptions
$\Rightarrow$ Instructions must reach the last stage of the pipeline in order.

Implementation of precise exceptions in the MIPS (2/4)

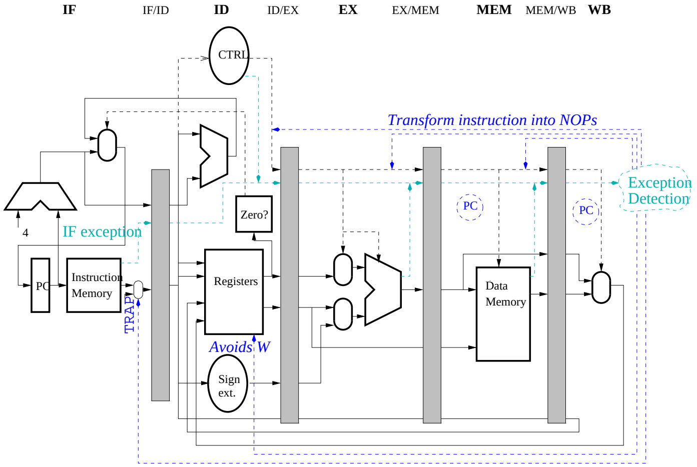
1. Each instruction entering the unit is related to a register with as many bits as stages in the pipeline able to raise an exception. Such register travels through the pipeline together with the instruction.
2. If an exception is raised, the register bit of the corresponding stage becomes "1". At the same time, the instruction generating the exception becomes a `NOP`
3. During the last stage, register's bits are checked
   If any bit is set, the following instructions become `NOP` and a `TRAP` is generated in the IF stage for the following cycle
4. The PC of the instruction raising the exception is stored. If the branch delay technique is used the PC of *delay_slot*+1 instructions must be also stored.
5. The exception handler takes the control.

Implementation of precise exceptions in the MIPS (3/4)

6. Once the handler(s) ends its execution, the PC (or PCs) is (are) restored, thus following the execution from that point.



Implementation of precise exceptions in the MIPS (4/4)


IF
IF/ID
ID
ID/EX
EX
EX/MEM
MEM
MEM/WB
WB
CTRL
Transform instruction into NOPs
Exception Detection
Zero?
PC
PC
IF exception
4
PC
Instruction Memory
TRAP
Registers
Data Memory
Avoids W
Sign ext.