Performance

- What does it mean when you say “system A is of a higher performance as compared to system B”?

Performance

- What does it mean when you say “system A is of a higher performance as compared to system B”?
- “Time taken to perform a certain task T on system A is less than that taken to perform the same task T on system B.”
  - Note that this statement is specific to the task T. For some other task T’, it may be possible that system B is the better performer.

Performance

```
int f = 1;
for(int i=2; i<=n; i++)
{
    f = f * i;
}
cout << f;
```

Program A

```
int f = 1;
for(int i=2; i<=2*n; i++)
{
    if (i <= n)
        f = f * i;
}
cout << f;
```

Program B

- System A: Program A is run on processor type P
- System B: Program B is run on processor type P
- Which system displays higher performance?

Performance

```
.main:
[0]     mov r2, 3
[4]     mov r1, 1
.loop:
[8]     mul r1, r1, r2
[12]    sub r2, r2, 1
[16]    cmp r2, 1
[20]    bgt .loop
[24]    st r1, 4[r0]
[28]    end
```

How many instructions are executed?

Performance

```
.main:
[0]    mov r2, 3
[4]    mov r1, 1
.loop:
[8]    mul r1, r1, r2
[12]   sub r2, r2, 1
[16]   cmp r2, 1
[20]   bgt .loop
[24]   st r1, 4[r0]
[28]   end
```

- Number of static instructions = 8
- Number of dynamic instructions = 12

Performance

```
int f = 1;
for(int i=2; i<=n; i++)
{
    f = f * i;
}
cout << f;
```

Program A

```
int f = 1;
for(int i=2; i<=2*n; i++)
{
    if (i <= n)
        f = f * i;
}
cout << f;
```

Program B

- System A: Program A is run on processor type P
- System B: Program B is run on processor type P
- System A displays higher performance because program A has fewer dynamic instructions (assuming every instruction takes equal amount of time)

SimpleRISC Processor Frequency

- How long does it take to execute a single instruction on our SimpleRISC processor?

SimpleRISC Processor Frequency

- How long does it take to execute a single instruction on our SimpleRISC processor?
- Work out the longest path in the combinational circuitry. Processor clock period should be greater than the time taken to execute this longest path. Let this clock period be t seconds. Processor frequency f = 1/t .
- Every instruction is executed in t s.
- Every instruction is executed in 1 cycle in our SimpleRISC processor.
  - Cycles per instruction, CPI = 1
  - Instructions per cycle, IPC = 1

Computing the Time a Program Takes

$$\begin{aligned}
\tau &= \#seconds \\
&= \frac{\#seconds}{\#cycles} * \frac{\#cycles}{\#instructions} * (\#instructions) \\
&= \underbrace{\frac{\#seconds}{\#cycles}}_{1/f} + \underbrace{\frac{\#cycles}{\#instructions}}_{CPI} * (\#instructions) \\
&= \frac{CPI * \#insts}{f}
\end{aligned}$$

* CPI ⟶ Cycles per instruction
* f ⟶ frequency (cycles per second)

The Performance Equation

$$P \propto \frac{IPC \ * f}{\#insts}$$

- IPC $\longrightarrow$ 1/CPI (Instructions per Cycle)
- What are the units of performance ?
  - **ANSWER** : arbitrary

Number of Instructions (#insts)

Static Instruction: The binary or executable of a program, contains a list of ***static instructions.***

Dynamic Instruction: A **dynamic instruction** is a running instance of a static instruction, which is created by the processor when an instruction enters the pipeline.

* Note that these are dynamic instructions
  * **NOT** static instructions
* A smart compiler can reduce the number of executed instructions

Number of Instructions(#insts) – 2

- Dead code removal
  - Often programmers write code that does not determine the final output
  - This code is redundant
  - It can be identified and removed by the **compiler**
- Function inlining
  - Very small functions have a lot of overhead → call, ret instructions, register spilling, and restoring
  - Paste the code of the callee in the code of the caller (known as **inlining**)

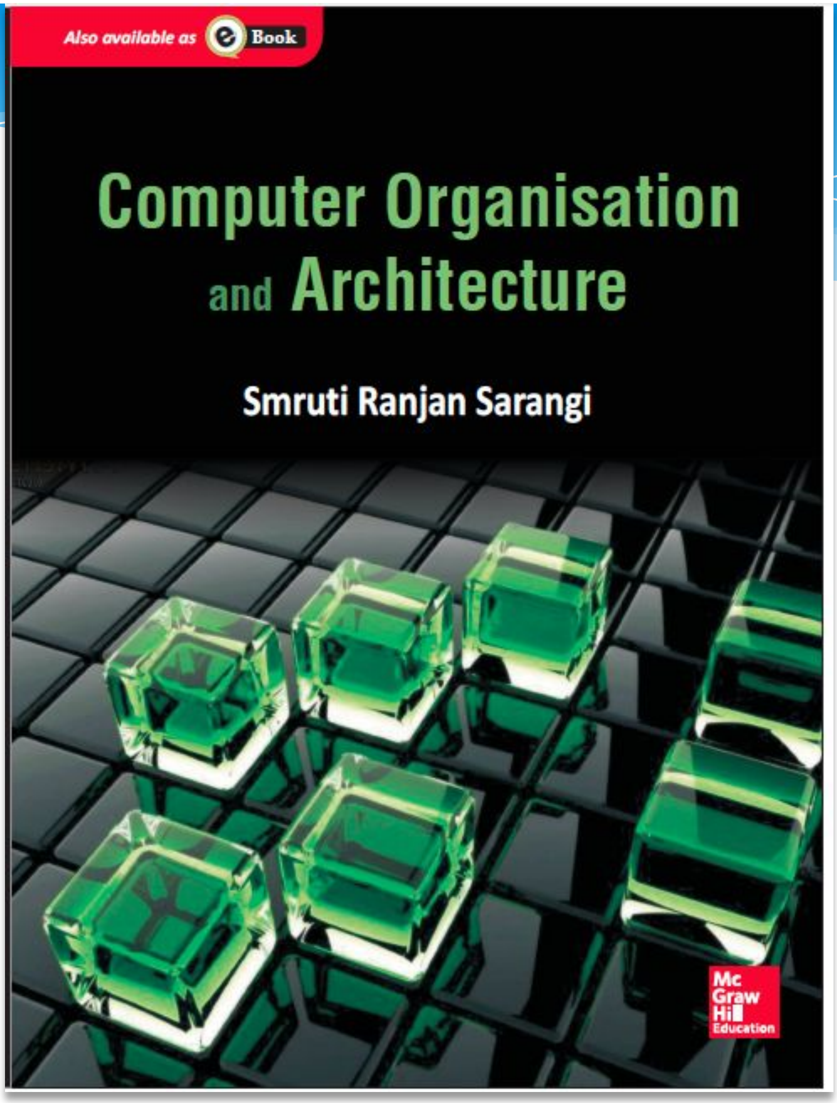
PowerPoint Slides

Computer Organisation and Architecture

Smruti Ranjan Sarangi,
IIT Delhi

Chapter 9 Principles of Pipelining

PROPRIETARY MATERIAL. © 2014 The McGraw-Hill Companies, Inc. All rights reserved. No part of this PowerPoint slide may be displayed, reproduced or distributed in any form or by any means, without the prior written permission of the publisher, or used beyond the limited distribution to teachers and educators permitted by McGraw-Hill for their individual course preparation. PowerPoint Slides are being provided only to authorized professors and instructors for use in preparing for classes using the affiliated textbook. No other use or distribution of this PowerPoint slide is permitted. The PowerPoint slide may not be sold and may not be distributed or be used by any student or any other third party. No part of the slide may be reproduced, displayed or distributed in any form or by any means, electronic or otherwise, without the prior written permission of McGraw Hill Education (India) Private Limited.



These slides are meant to be used along with the book: Computer Organisation and Architecture, Smruti Ranjan Sarangi, McGrawHill 2015
Visit: http://www.cse.iitd.ernet.in/~srsarangi/archbooksoft.html

Outline

* Overview of Pipelining
* A Pipelined Data Path
* Pipeline Hazards
* Pipeline with Interlocks
* Forwarding
* Performance Metrics
* Interrupts/ Exceptions

Up till now ….

* We have designed a processor that can execute all the SimpleRisc Instructions
* We have look at two styles :
  * With a hardwired control unit
  * Microprogrammed control unit
    * Microprogrammed data path
    * Microassembly Language
    * Microinstructions

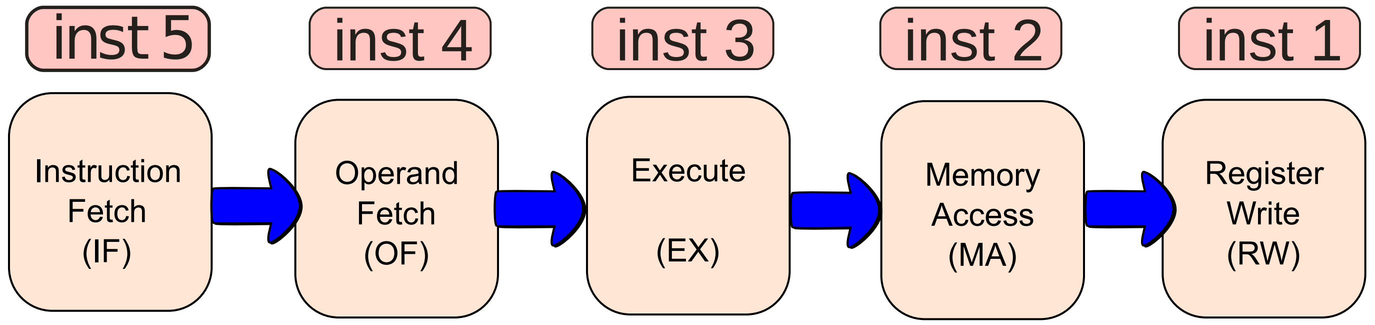
Designing Efficient Processors

- Microprogrammed processors are much slower that hardwired processors
- Even hardwired processors
  - Have a lot of waste !!!
  - We have 5 stages.
  - What is the IF stage doing, when the MA stage is active ?
    - **ANSWER** : It is **idling**

The Notion of Pipeling

* Let us go back to the car assembly line
  * Is the engine shop idle, when the paint shop is painting a car ?
  * **NO** : It is building the engine of another car
  * When this engine goes to the body shop, it builds the engine of another car, and so on ….
  * **Insight** :
    * Multiple cars are built at the same time.
    * A car proceeds from one stage to the next

Pipelined Processors



* The IF, ID, EX, MA, and RW stages process 5 instructions simultaneously
* Each instruction proceeds from one stage to the next
* This is known as pipelining

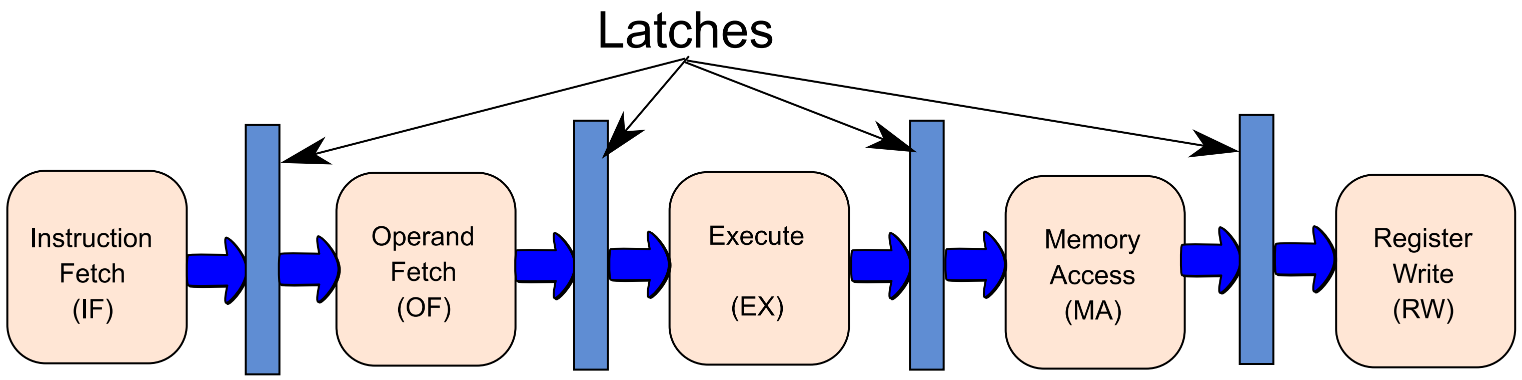
Advantages of Pipelining

* We keep all parts of the data path, busy all the time
* Let us assume that all the 5 stages do the same amount of **work**
  * Without pipelining, every T seconds, an instruction completes its execution
  * With pipelining, every T/5 seconds, a new instruction completes its execution

Design of a Pipeline

- Splitting the Data Path
  - We divide the data path into 5 parts : IF, OF, EX, MA, and RW
- Timing
  - We insert latches (registers) between consecutive stages
  - 4 Latches $\longrightarrow$ IF-OF, OF-EX, EX-MA, and MA-RW
  - At the negative edge of a clock, an instruction moves from one stage to the next

Pipelined Data Path with Latches



- Add a latch between subsequent stages.
  - Triggered by a negative clock edge

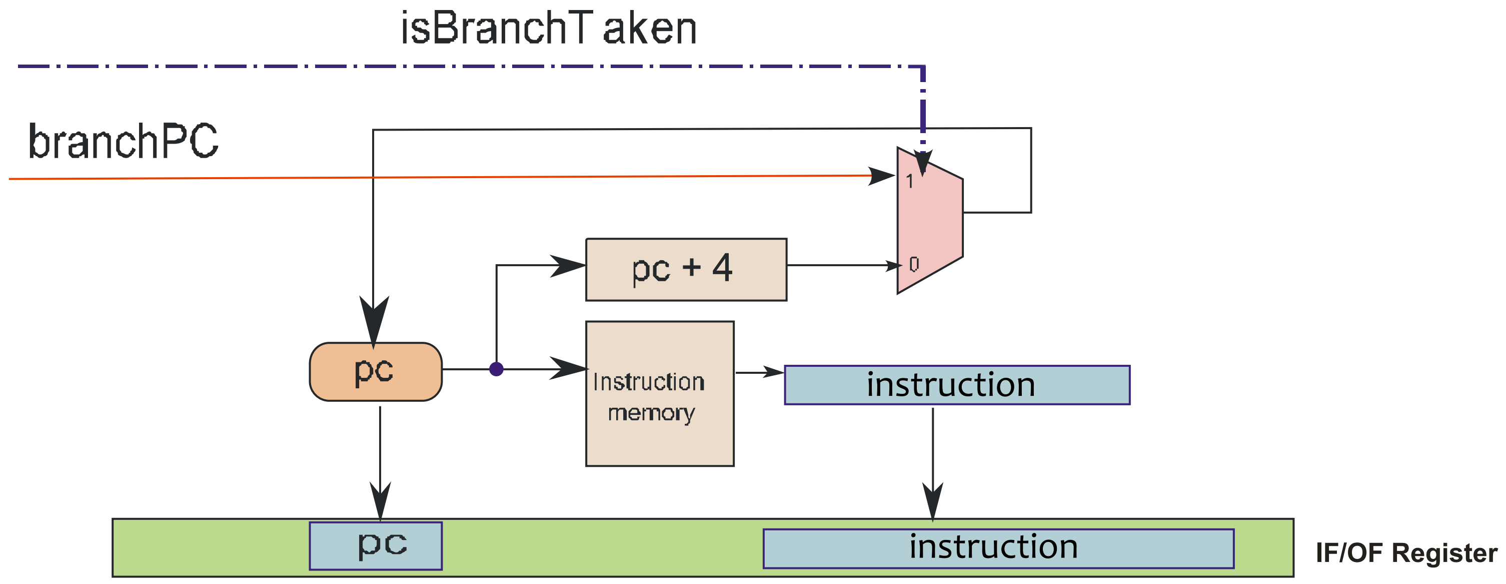
The Instruction Packet

* What travels between stages ?
  * **ANSWER** : the instruction packet
* Instruction Packet
  * Instruction contents
  * Program counter
  * All intermediate results
  * Control signals
* Every instruction moves with its entire state, no interference between instructions

Outline

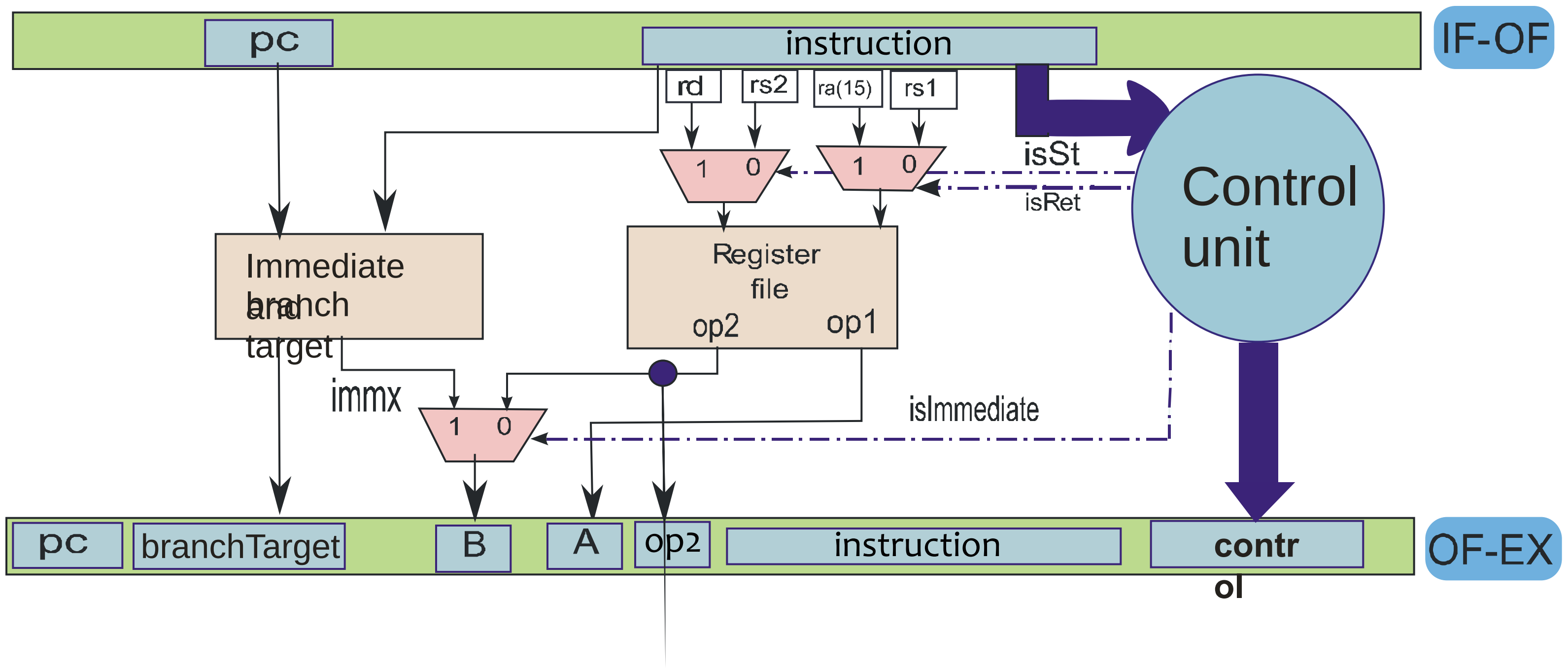
* Overview of Pipelining
* A Pipelined Data Path
* Pipeline Hazards
* Pipeline with Interlocks
* Forwarding
* Performance Metrics
* Interrupts/ Exceptions

IF Stage



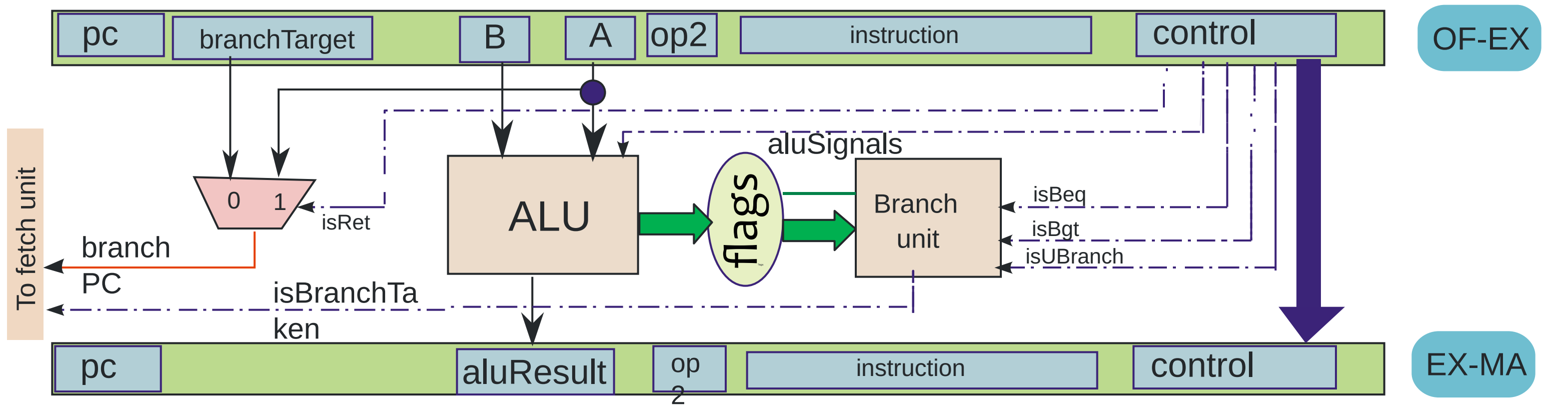
* Instruction contents saved in the instruction field

OF Stage



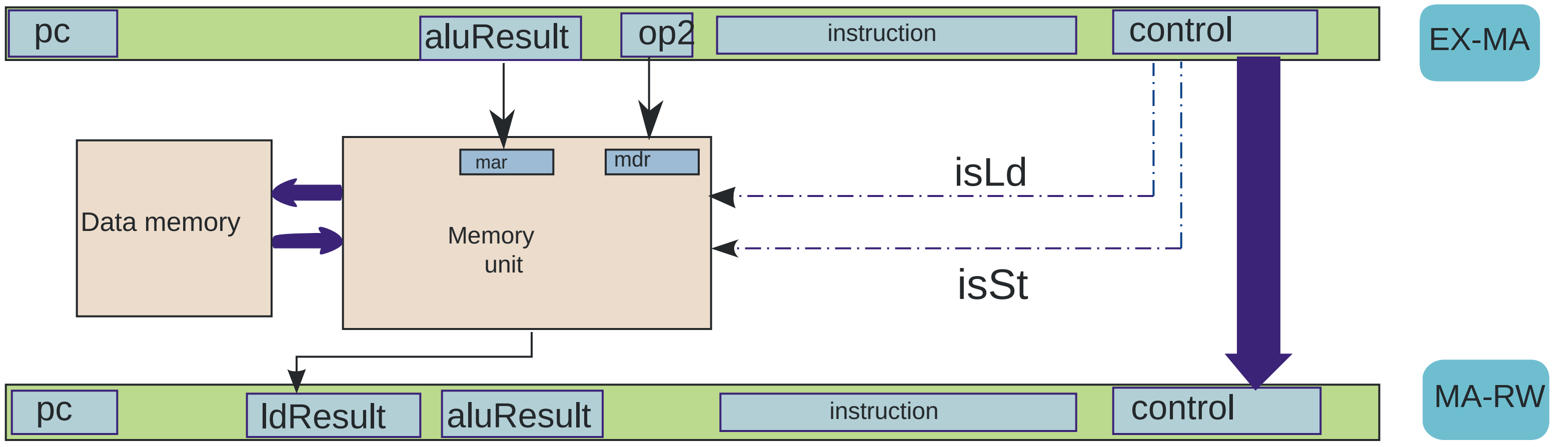
* A, B → ALU Operands, op2 (store operand), control (set of all control signals)

EX Stage



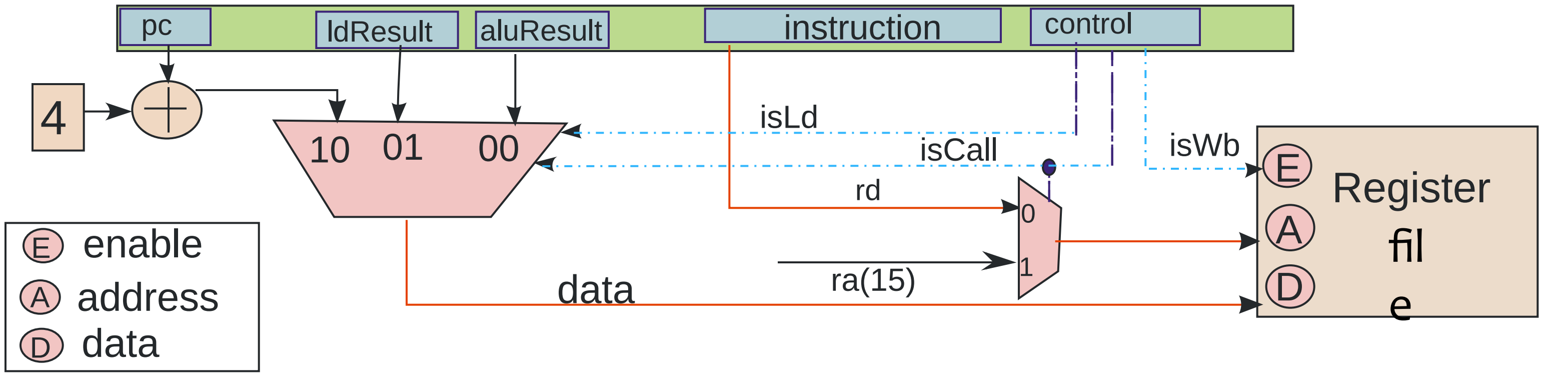
* aluResult → result of the ALU Operation
* op2, control, pc, instruction (passed from OF-EX)

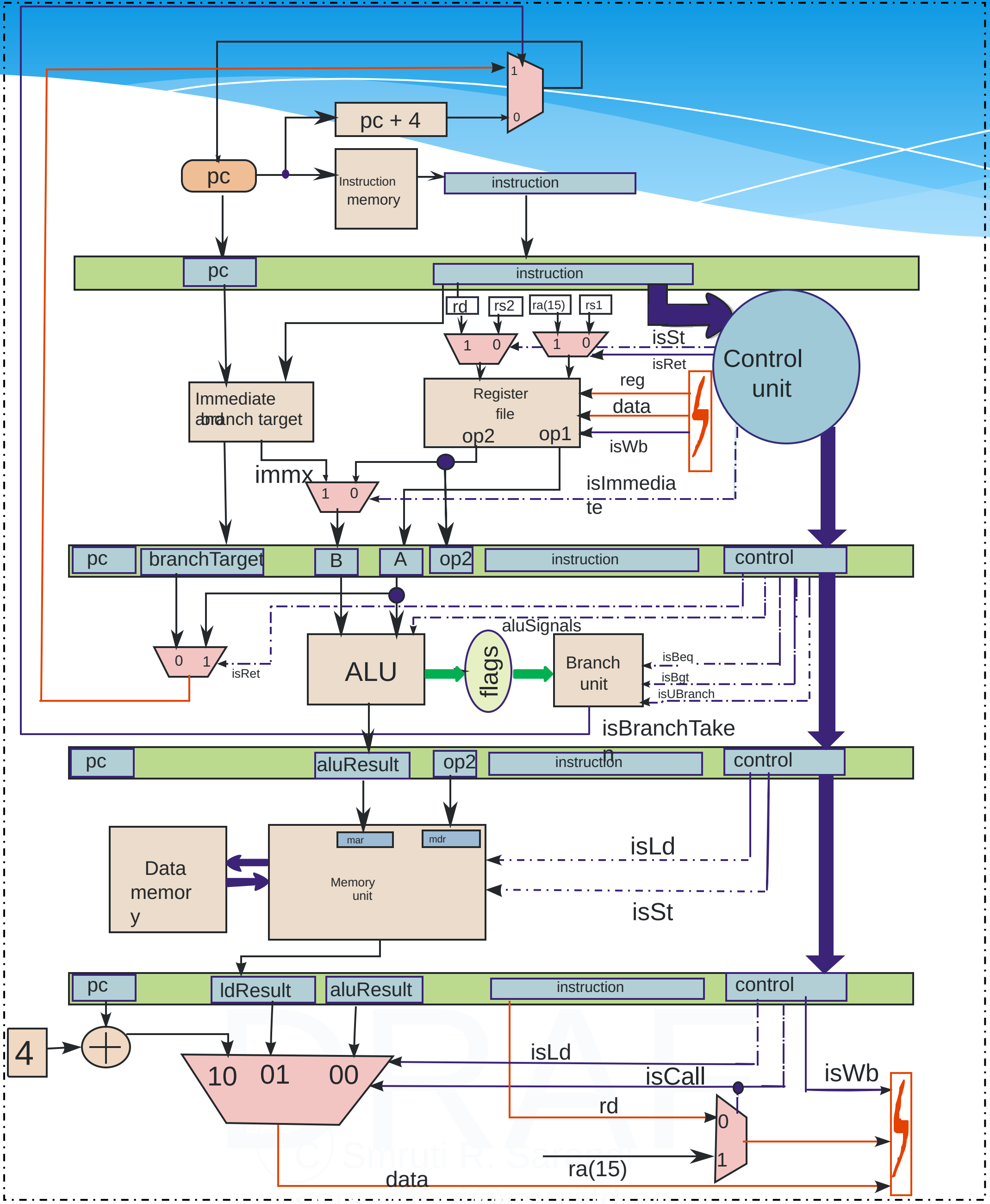
MA Stage



* ldResult → result of the load operation
* aluResult, control, pc, instruction (passed from EX-MA)

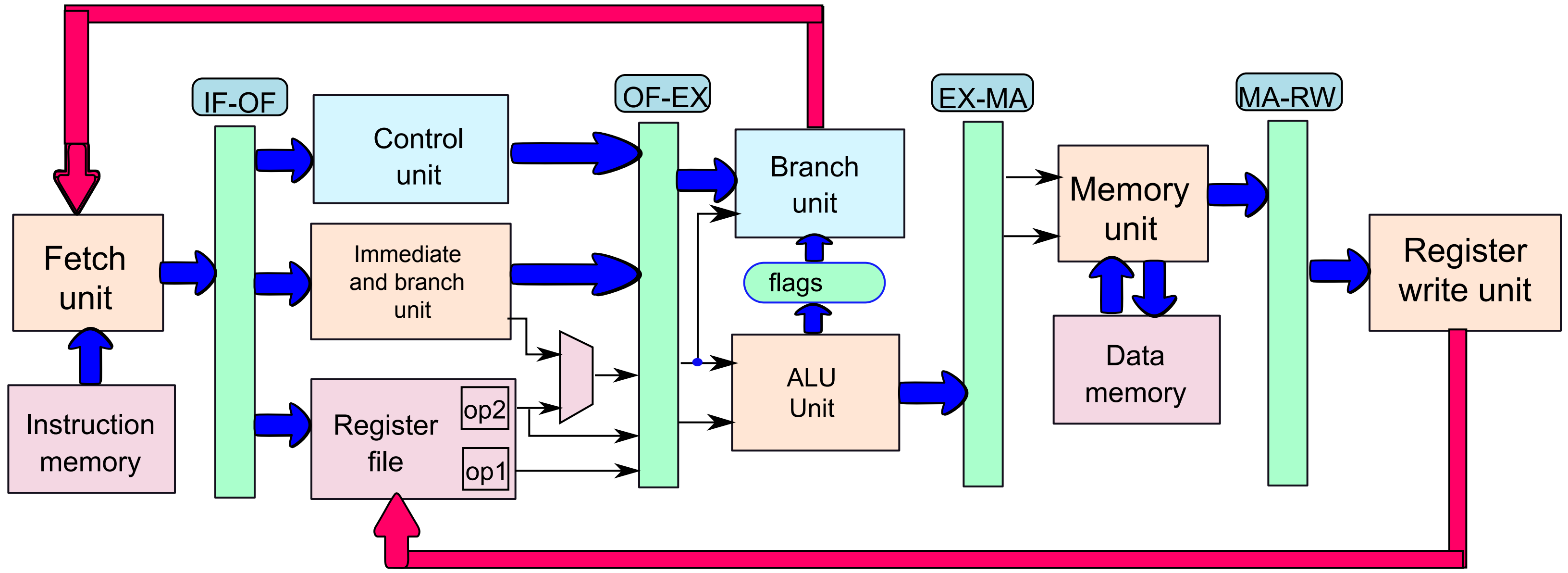
RW Stage


pc
ldResult
aluResult
instruction
control
4
10
01
00
isLd
isCall
rd
isWb
0
1
ra(15)
data
E
A
D
Register
fil
e
E enable
A address
D data



pc + 4
1
0
pc
Instruction memory
instruction
pc
instruction
rd
rs2
ra(15)
rs1
1 0
1 0
isSt
isRet
Control unit
reg
data
isWb
Register file
op2
op1
Immediate and branch target
immx
1 0
isImmediate
pc
branchTarget
B
A
op2
instruction
control
aluSignals
0 1
isRet
ALU
flags
Branch unit
isBeq
isBgt
isUBranch
isBranchTaken
pc
aluResult
op2
instruction
control
mar
mdr
Memory unit
Data memory
isLd
isSt
pc
ldResult
aluResult
instruction
control
4
10 01 00
isLd
isCall
isWb
rd
0
1
ra(15)
data

Abridged Diagram



Outline

* Overview of Pipelining
* A Pipelined Data Path
* Pipeline Hazards
* Pipeline with Interlocks
* Forwarding
* Performance Metrics
* Interrupts/ Exceptions

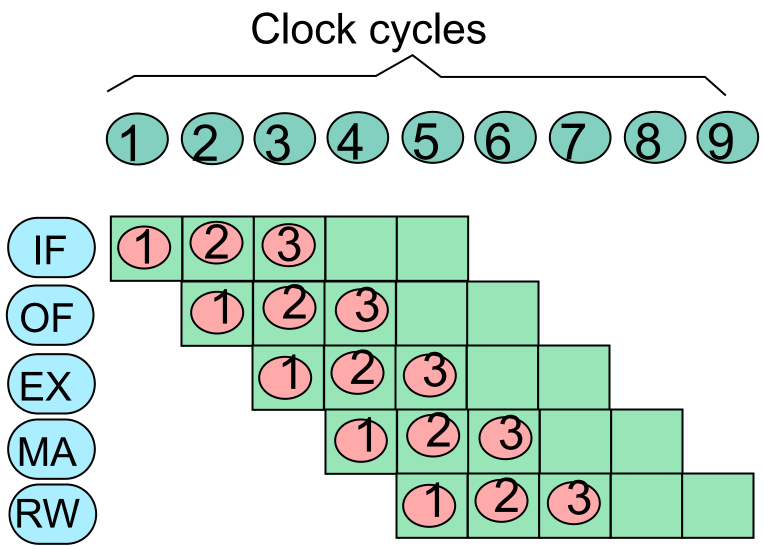
Pipeline Hazards

- Now, let us consider correctness
- Let us introduce a new tool → Pipeline Diagram

[1]: add r1, r2, r3

[2]: sub r4, r5, r6

[3]: mul r8, r9, r10



Rules for Constructing a Pipeline Diagram

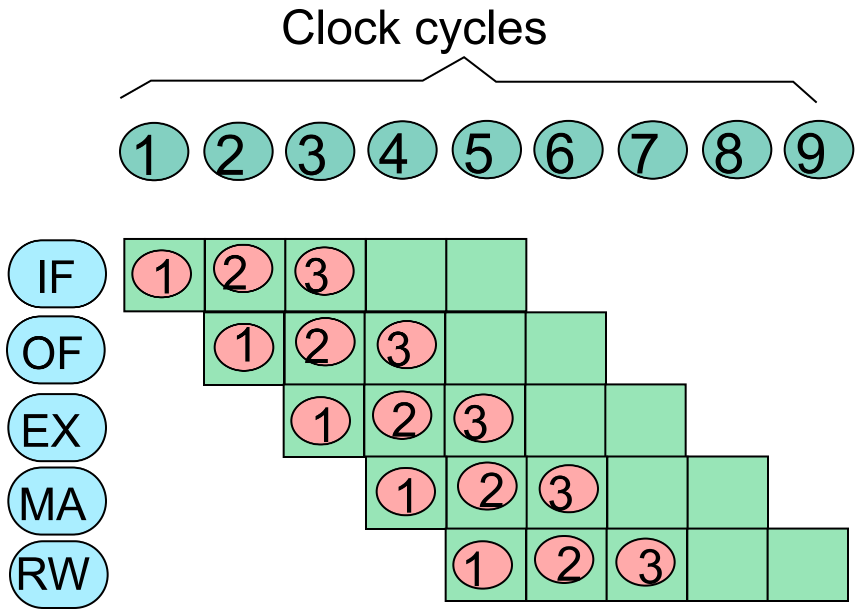
* It has 5 rows
  * One per each stage
  * The rows are named : IF, OF, EX, MA, and RW
* Each column represents a clock cycle
* Each cell represents the execution of an instruction in a stage
  * It is annotated with the name(label) of the instruction
* Instructions proceed from one stage to the next across clock cycles

Example

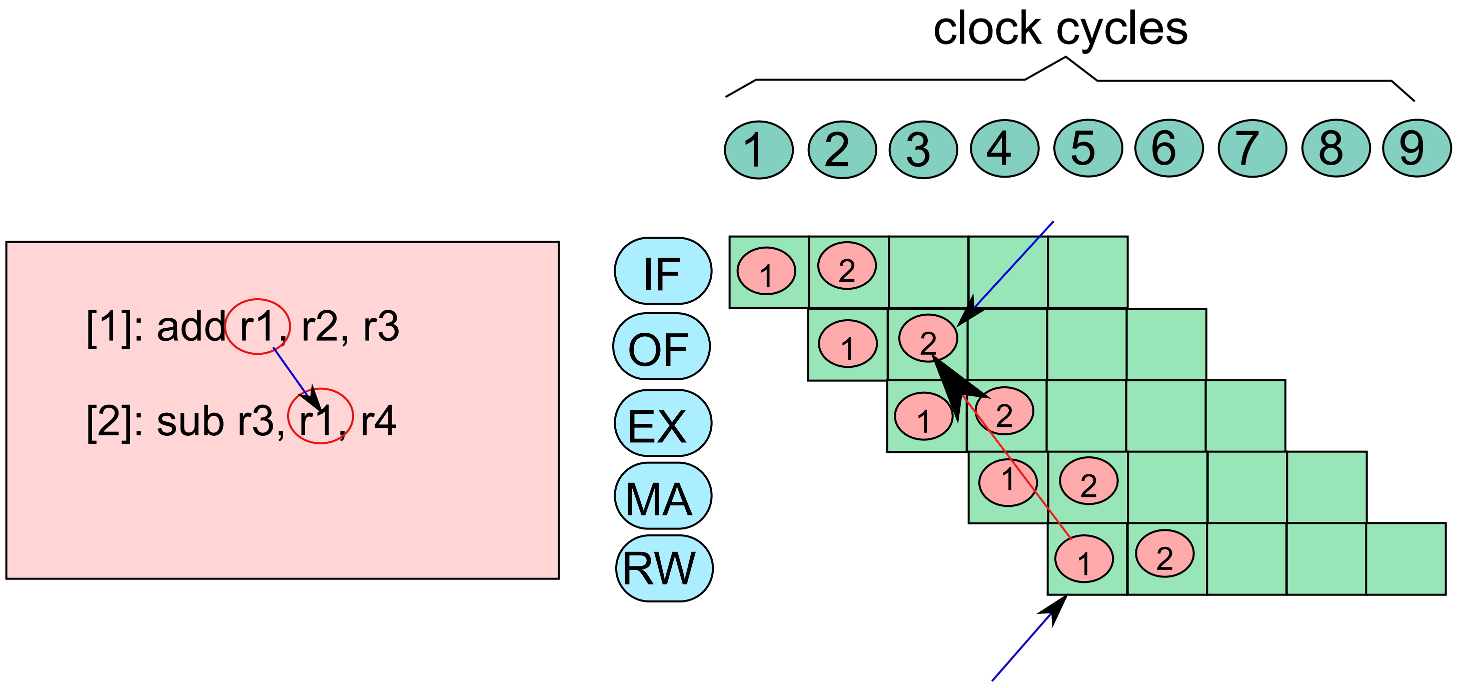
[1]: add r1, r2, r3

[2]: sub r4, r2, r5

[3]: mul r5, r8, r9



Data Hazards



* Instruction 2 will read incorrect values !!!

Data Hazard

Definition: A hazard is defined as the possibility of erroneous execution of an instruction in a pipeline. A data hazard represents the possibility of erroneous execution because of the unavailability of data, or the availability of incorrect data.

* This situation represents a data hazard
* In specific,
  * it is a RAW (read after write) hazard
* The earliest we can dispatch instruction 2, is cycle 5

Other Types of Data Hazards

* Our pipeline is in-order

Definition: In an in-order pipeline (such as ours), a preceding instruction is always ahead of a succeeding instruction in the pipeline. Modern processors however use out-of-order pipelines that break this rule. It is possible for later instructions to execute before earlier instructions.

* We will only have RAW hazards in our pipeline.
* Out-of-order pipelines can have WAR and WAW hazards

WAW Hazards

```
[1]: add r1, r2, r3
[2]: sub r1, r4, r3
```

* Instruction [2] cannot write the value of r1, before instruction [1] writes to it, will lead to a WAW hazard

WAR Hazards

```
[1]: add r1, r2, r3
[2]: add r2, r5, r6
```

* Instruction [2] cannot write the value of r2, before instruction [1] reads it $\longrightarrow$ will lead to a WAR hazard

```
[1] add r1, r2, r3
[2] ld r4, 8[r1]
[3] sub r5, r5, r2
[4] st r4, 12[r2]
[5] sub r6, r4, r3
[6] add r7, r4, r3
```

Q. How many dependencies?

Q. How many dependencies require conflict resolution?

Control Hazards

```
[1]: beq .foo
[2]: mov r1, 4
[3]: add r2, r4, r3
...
...
.foo:
[100]: add r4, r1, r2
```

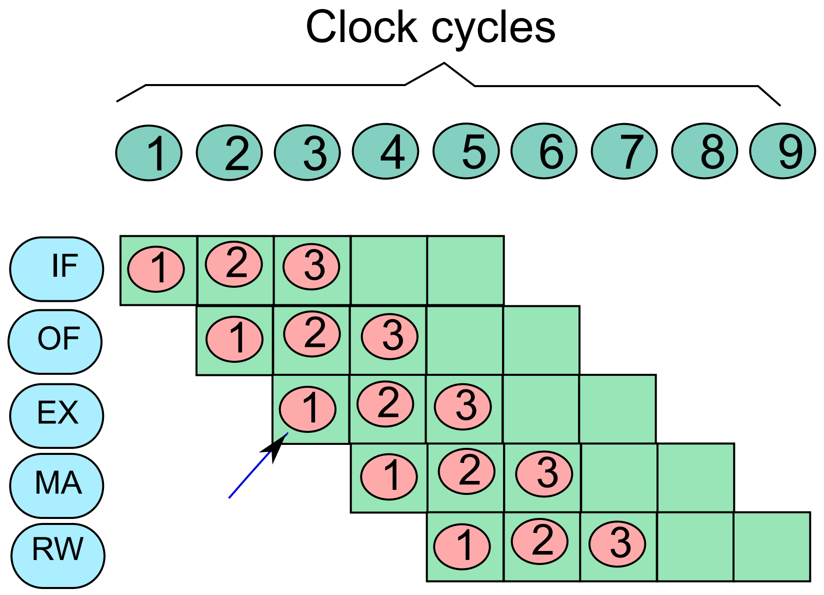
* If the branch is taken, instructions [2] and [3], might get fetched, incorrectly

Control Hazard – Pipeline Diagram

```
[1]: beq .foo

[2]: mov r1, 4

[3]: add r2, r4, r3
```



* The two instructions fetched immediately after a branch instruction might have been fetched incorrectly.

```
[1] mov r1, 0
[2] mov r2, 0
[3] cmp r1, r2
[4] beq .foo
[5] sub r6, r4, r3
[6] add r7, r4, r3
[7] mul r8, r4, r3
…
…
.foo
[100] add r5, r4, r1
```

Control Hazards

- The two instructions fetched immediately after a branch instruction might have been fetched incorrectly.
- These instructions are said to be on the wrong path
- A control hazard represents the possibility of erroneous execution in a pipeline because instructions in the wrong path of a branch can possibly get executed and save their results in memory, or in the register file

Structural Hazards

- A structural hazard may occur when two instructions have a conflict on the same set of resources in a cycle
- Example :
  - Assume that we have an add instruction that can read one operand from memory
  - add r1, r2, 10[r3]

Structural Hazards - II

```
[1]: st r4, 20[r5]
[2]: sub r8, r9, r10
[3]: add r1, r2, 10[r3]
```

* This code will have a structural hazard
  * [3] tries to read 10[r3] (MA unit) in cycle 4
  * [1] tries to write to 20[r5] (MA unit) in cycle 4
* Does not happen in our pipeline

Solutions in Software

* Data hazards
  * Insert nop instructions, reorder code

```
[1]: add r1, r2, r3
[2]: sub r3, r1, r4
```

↓

```
[1]: add r1, r2, r3
[2]: nop
[3]: nop
[4]: nop
[5]: sub r3, r1, r4
```

McGraw Hill Education

Code Reordering

```
add r1, r2, r3
add r4, r1, 3
add r8, r5, r6
add r9, r8, r5
add r10, r11, r12
add r13, r10, 2
```

Code Reordering

```
add r1, r2, r3
add r4, r1, 3
add r8, r5, r6
add r9, r8, r5
add r10, r11, r12
add r13, r10, 2
```

```
add r1, r2, r3
add r8, r5, r6
add r10, r11, r12
nop
add r4, r1, 3
add r9, r8, r5
add r13, r10, 2
```

Control Hazards

* Trivial Solution : Add two nop instructions after every branch
* Better solution :
  * Assume that the two instructions fetched after a branch are valid instructions
  * These instructions are said to be in the delay slots
  * Such a branch is known as a delayed branch

Example with 2 Delay Slots

```
add r1, r2, r3
add r4, r5, r6
b .foo
add r8, r9, r10
```

```
b .foo
add r1, r2, r3
add r4, r5, r6
add r8, r9, r10
```

* The compiler transfers instructions before the branchto the delay slots.

* If it cannot find 2 valid instructions, it inserts nops.

Outline

* Overview of Pipelining
* A Pipelined Data Path
* Pipeline Hazards
* Pipeline with Interlocks
* Forwarding
* Performance Metrics
* Interrupts/ Exceptions

Why interlocks ?

* We cannot always trust the compiler to do a good job, or even introduce nop instructions correctly.
* Compilers now need to be tailored to specific hardware.
* We should ideally not expose the details of the pipeline to the compiler (might be confidential also)
* Hardware mechanism to enforce correctness ⟶ **interlock**

Two kinds of Interlocks

* Data-Lock
  * Do not allow a consumer instruction to move beyond the OF stage till it has read the correct values. Implication : **Stall** the IF and OF stages.
* Branch-Lock
  * We never execute instructions in the wrong path.
* The hardware needs to ensure both these conditions.

Comparison between Software and Hardware

Attribute	Software	Hardware(withinterlocks)
Portability	Limited to a specific processor	Programs can be run on any processor irrespective of the nature of the pipeline
Branches	Possible to have no performance penalty, by using delay slots	Need to stall the pipeline for 2 cycles in our design
RAW hazards	Possible to eliminate them through code scheduling	Need to stall the pipeline
Performance	Highly dependent on the nature of the program	The basic version of a pipeline with interlocks is expected to be slower than the version that relies on software

Conceptual Look at Pipeline with Interlocks

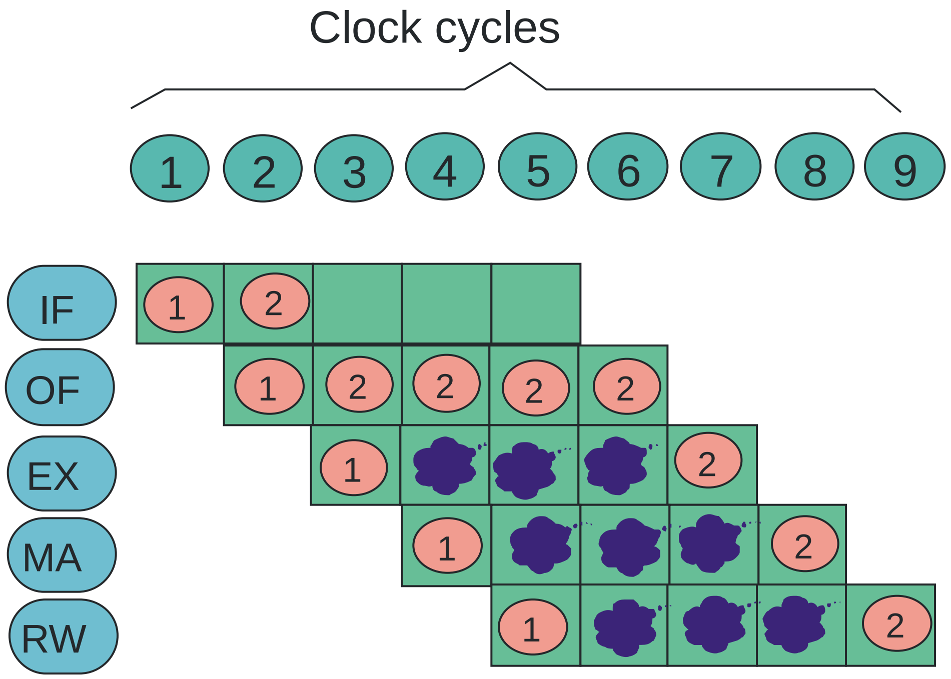
```
[1]: add r1, r2, r3
[2]: sub r4, r1, r2
```

* We have a RAW hazard
* We need to stall, instruction [2] at the OF stage for 3 cycles.
* We need to keep sending nop instructions to the EX stage during these 3 cycles

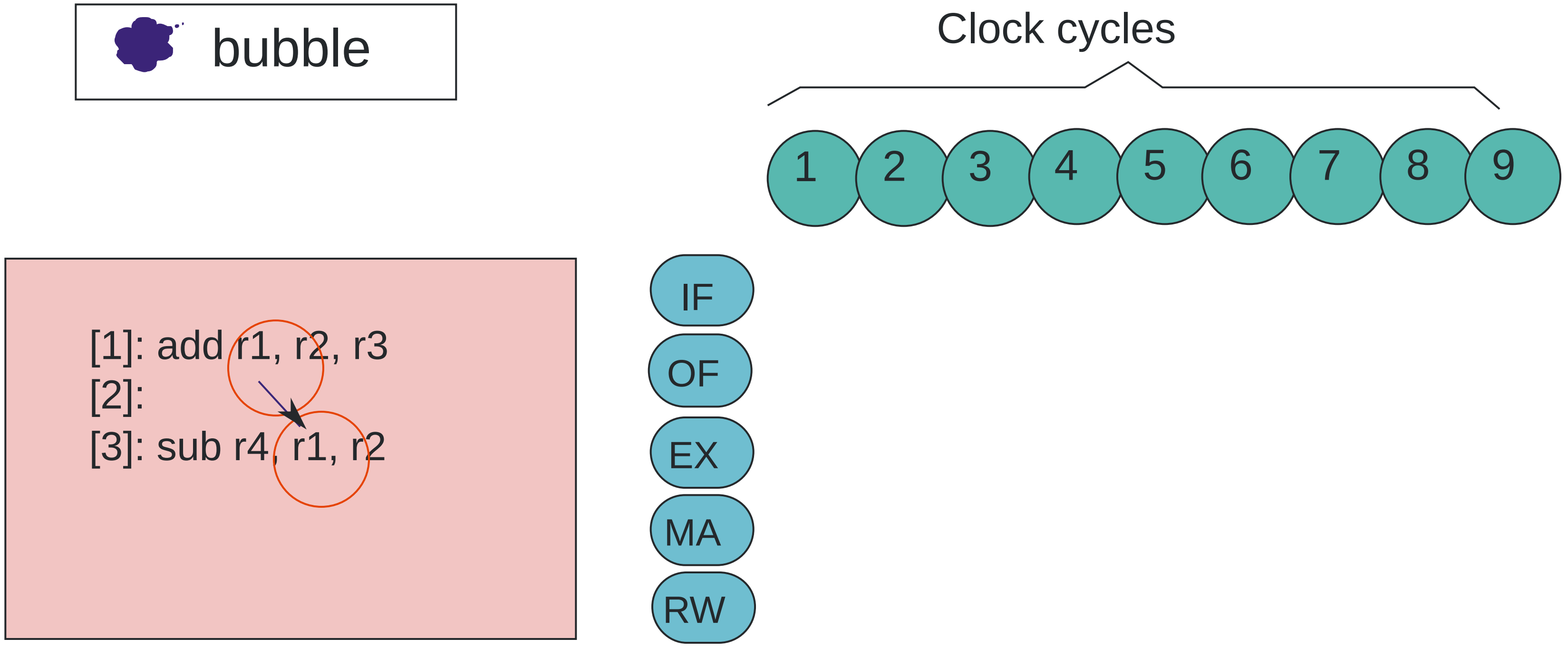
Example

[1]: add r1, r2, r3

[2]: sub r4, r1, r2



Example



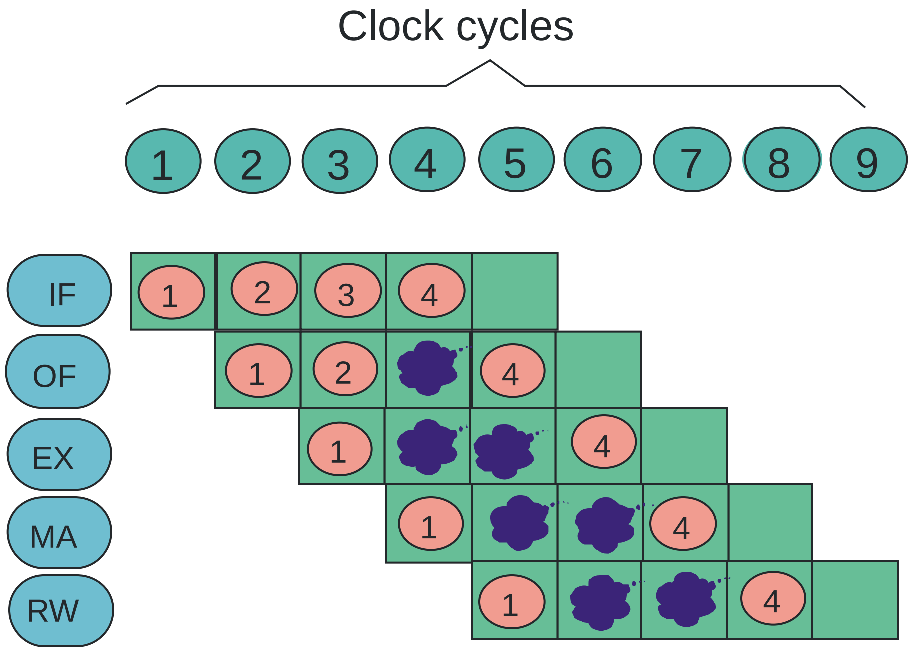
```
[1]: add r1, r2, r3
[2]:
[3]: sub r4, r1, r2
```

A Pipeline Bubble

* A pipeline bubble is inserted into a stage, when the previous stage needs to be stalled
* It is a nop instruction
* To insert a bubble
  * Create a nop instruction packet
  * OR, Mark a designated bubble bit to 1

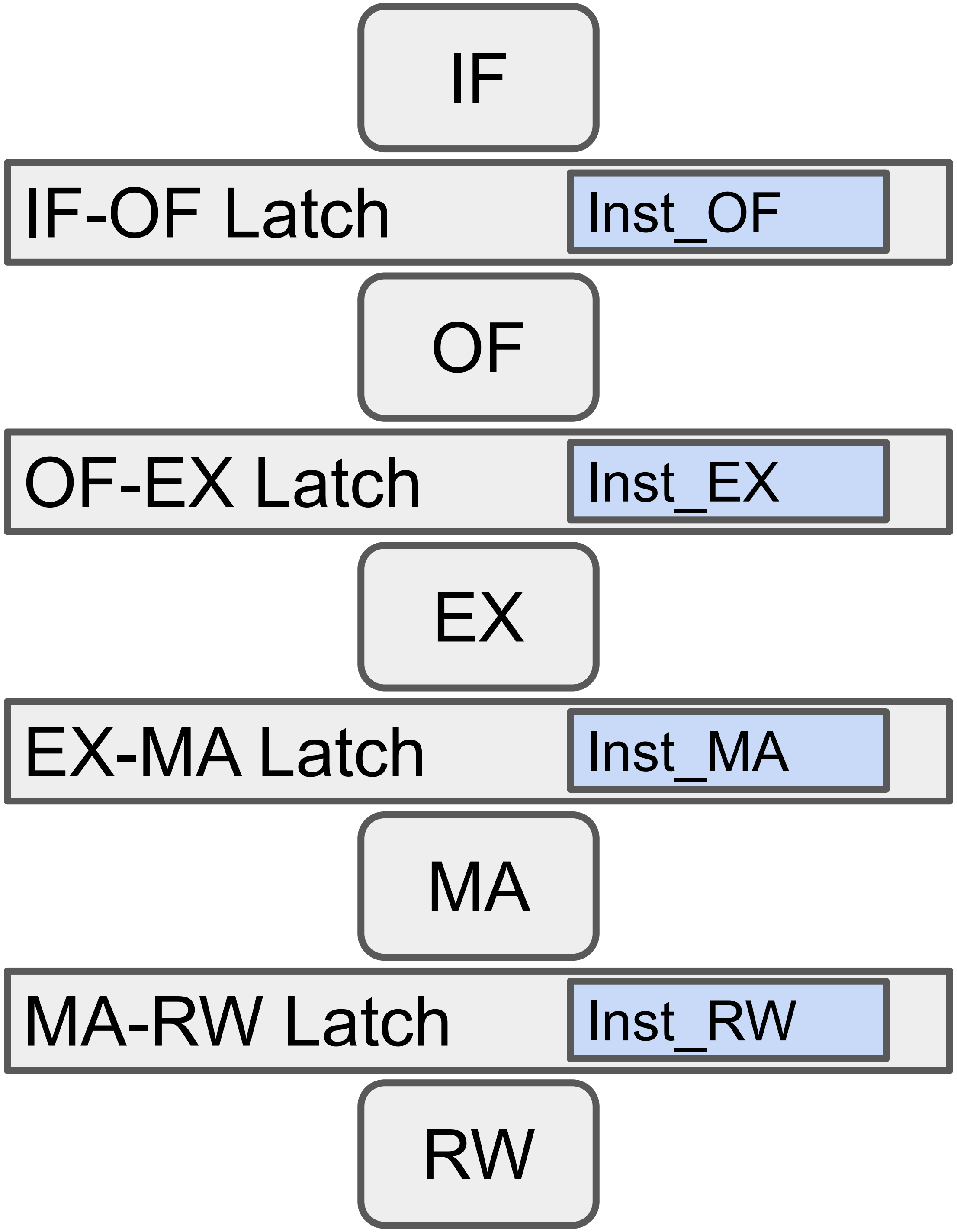
Bubbles in the Case of a Branch Instruction

```
[1]: beq. foo
[2]: add r1, r2, r3
[3]: sub r4, r5, r6
....
....
.foo:
[4]: add r8, r9, r10
```



Control Hazards and Bubbles

- We know that an instruction is a branch in the OF stage
- When it reaches the EX stage and the branch is taken, let us convert the instructions in the IF, and OF stages to bubbles
- Ensures the branch-lock condition


IF
IF-OF Latch
Inst_OF
OF
OF-EX Latch
Inst_EX
EX
EX-MA Latch
Inst_MA
MA
MA-RW Latch
Inst_RW
RW

Ensuring the Data-Lock Condition

- When an instruction reaches the OF stage, check if it has a conflict with any of the instructions in the EX, MA, and RW stages
- If there is **no** conflict, nothing needs to be done
- Otherwise, stall the pipeline (IF and OF stages only)

Algorithm 5: Algorithm to detect conflicts between instructions

```
Data: instructions, [A], and [B]
Result: conflict exists (true), no conflict (false)
if [A].opcode ∈ (nop,b,beq,bgt,call) then
        /* Does not read from any register    */
    return false
end
if [B].opcode ∈ (nop, cmp, st, b, beq, bgt, ret) then
    /* Does not write to any register       */
    return false
end
    /* Set the sources  */
src1 ← [A].rs1
src2 ← [A].rs2
if [A].opcode = st then
    src2 ← [A].rd
end
if [A].opcode = ret then
    src1 ← ra
end
hasSrc1 ← true
 if ([A] ∈ (not, mov)) hasSrc1 ← false
```

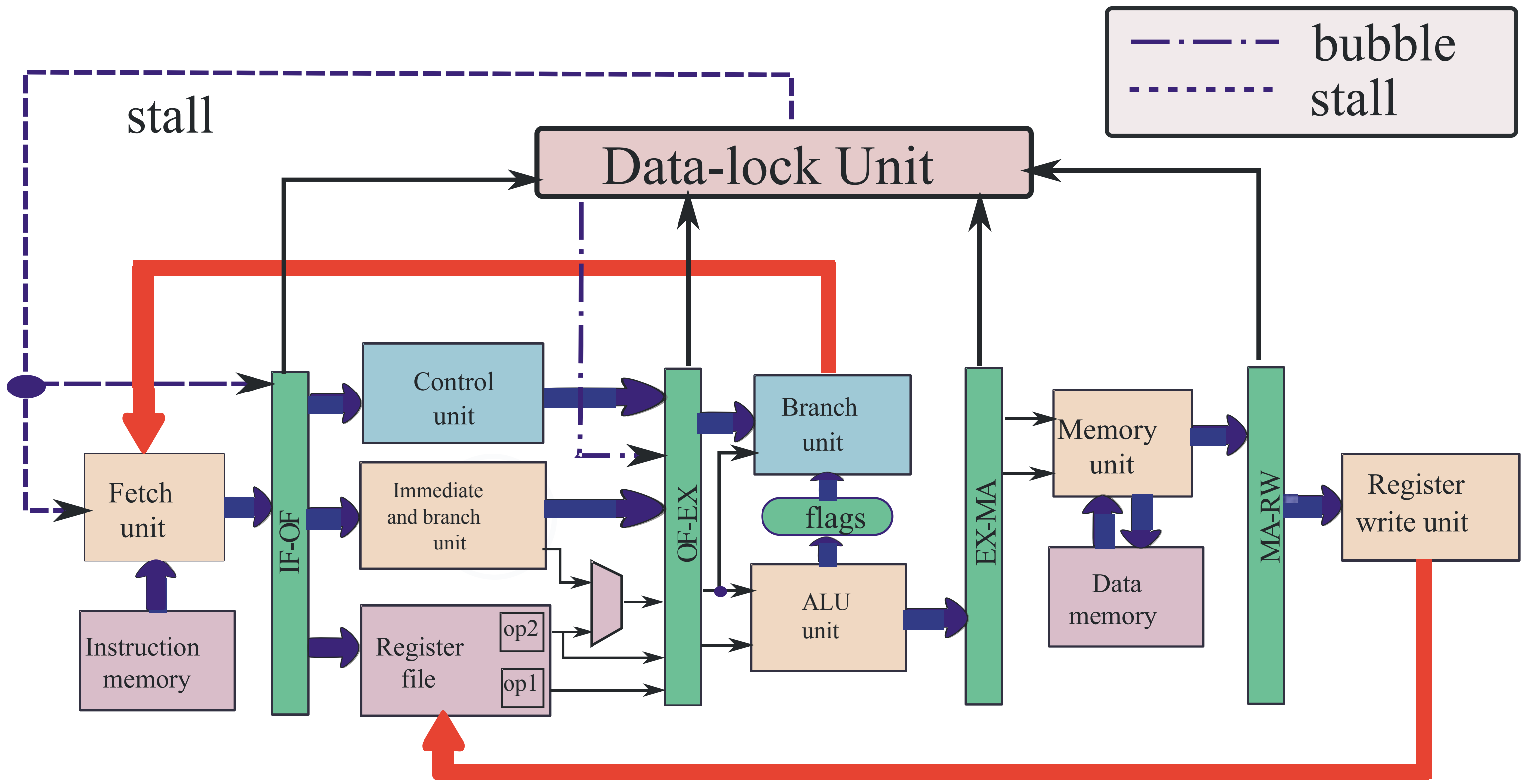
```
dest ← [B].rd
if [B].opcode = call then
    dest ← ra
end
/* Check the second operand to see if it is a register      */
hasSrc2 ← true
if [A].opcode  ≠ ( st) then
    if [A].I = 1 then
        hasSrc2 ← false
    end
end
/* Detect conflicts  */      */
if  (hasSrc1 = true) and (src1 = dest) then
    return true
end
else if (hasSrc2 = true) and (src2 = dest) then
    return true
end
return false
```

How to Stall a Pipeline ?

- Disable the write functionality of :
  - The IF-OF register
  - and the Program Counter (PC)
- To insert a bubble
  - Write a bubble (nop instruction) into the OF-EX register

Mc Graw Hill Education

Data Path with Interlocks (Data-Lock)



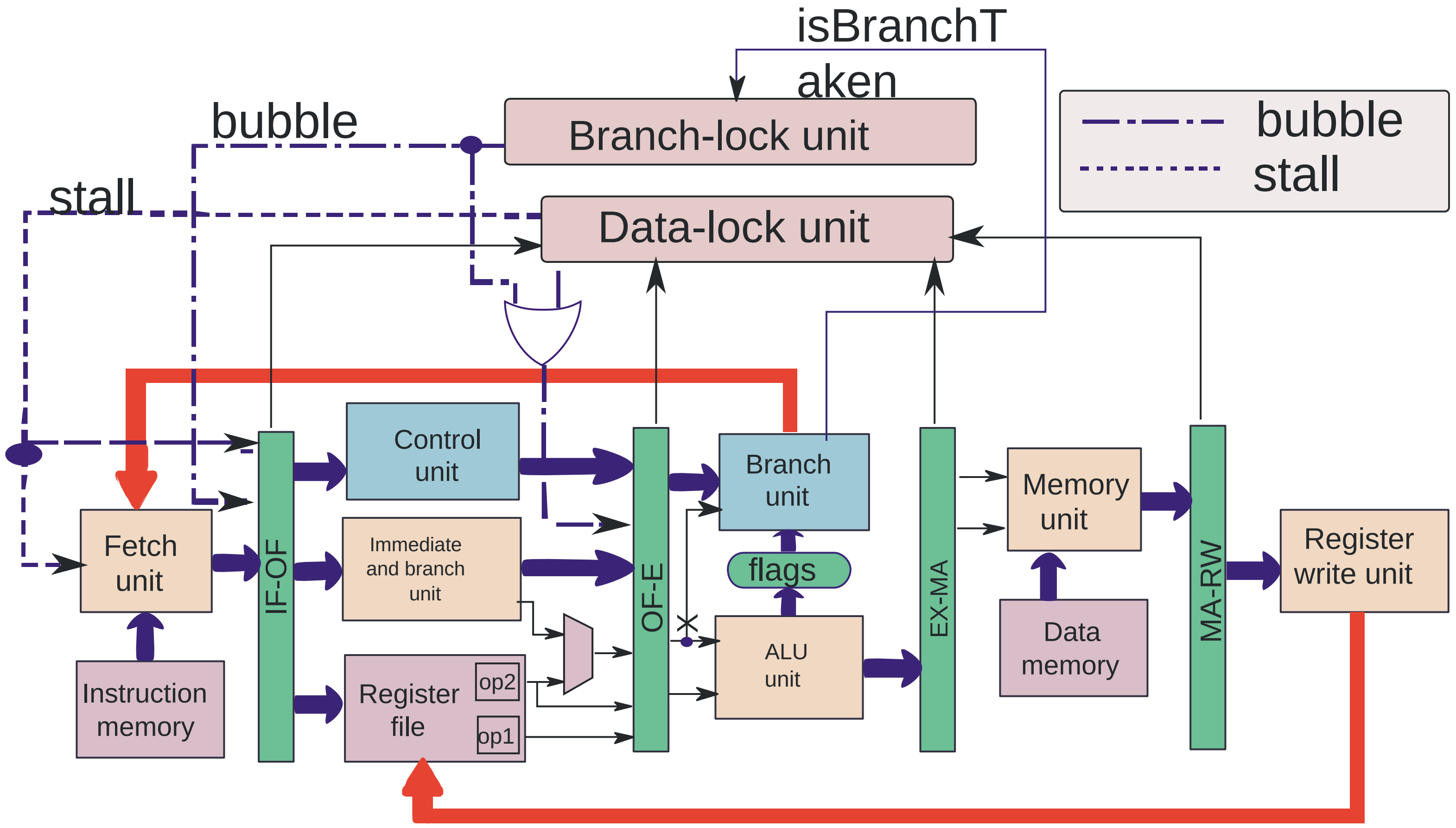
Ensuring the Branch-Lock Condition

- Option 1 :
  - Use delay slots (interlocks not required)
- Option 2 :
  - Convert the instructions in the IF, and OF stages, to bubbles once a branch instruction reaches the EX stage.
  - Start fetching from the next PC (not taken) or the branch target (taken)

Ensuring the Branch-Lock Condition - II

* Option 3
  * If the branch instruction in the EX stage is **taken**, then invalidate the instructions in the IF and OF stages. Start fetching from the branch target.
  * Otherwise, do not take **any special action**
  * **This method is also called predict not-taken (we shall use this method because it is more efficient that option 2)**

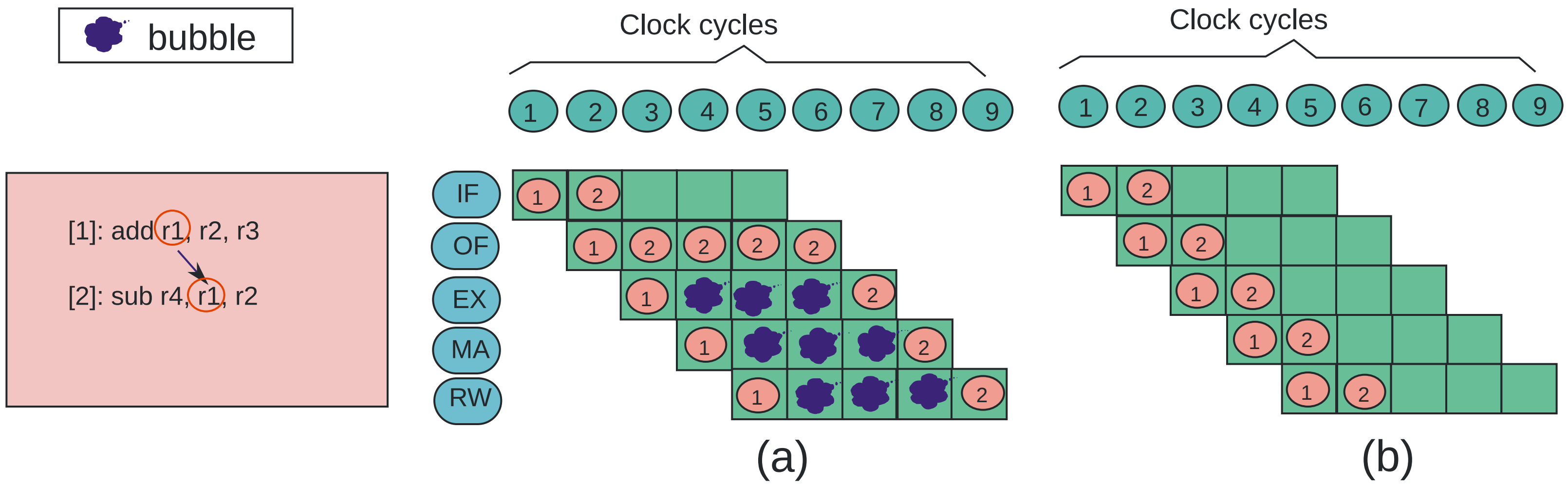
Data Path with Interlocks



Outline

* Overview of Pipelining
* A Pipelined Data Path
* Pipeline Hazards
* Pipeline with Interlocks
* Forwarding
* Performance Metrics
* Interrupts/ Exceptions

Relook at the Pipeline Diagram

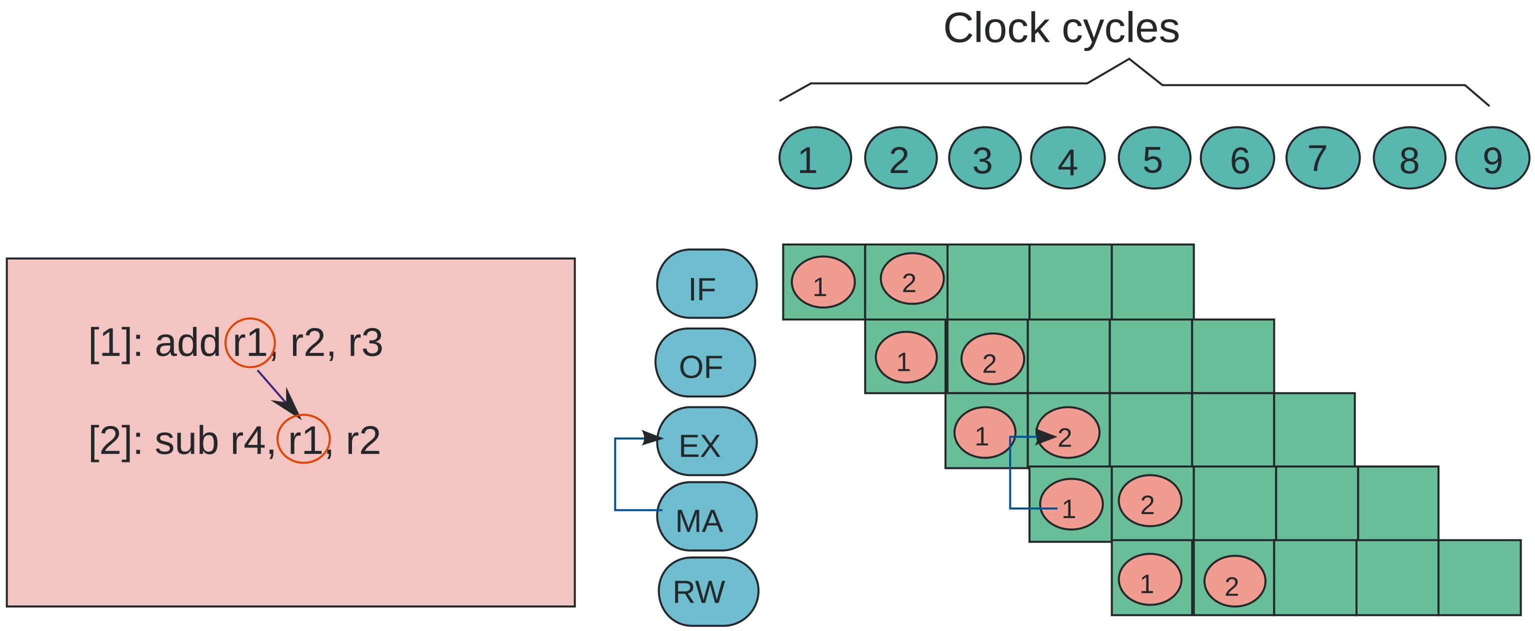


- (a) ⟶ with bubbles
- (b) ⟶ no bubbles (may lead to wrong results)

Crucial Insight (Figure (b))

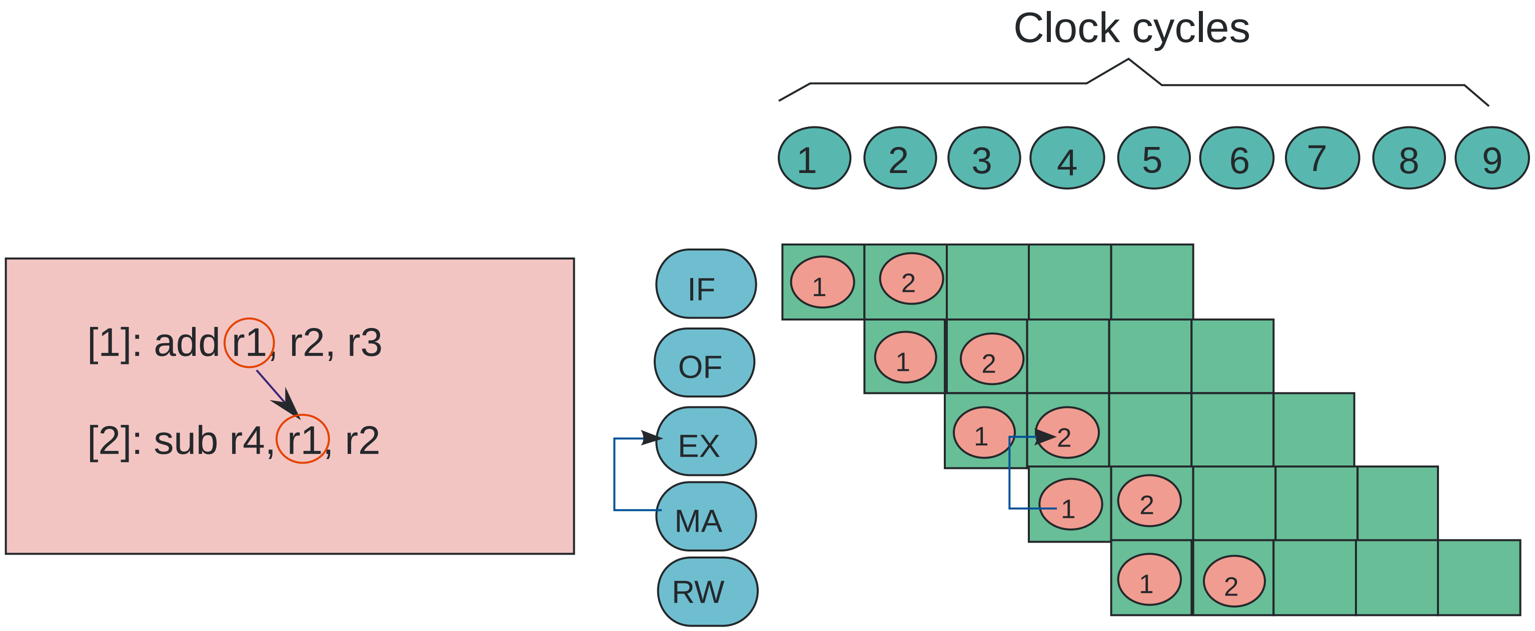
* When does instruction 2 need the value of r1 ?
    * ANSWER : Cycle 3, OF Stage (wrong!!!)
    * **CORRECT ANSWER** : Cycle 4, EX Stage
* When does instruction 1 produce the value of r1 ?
    * ANSWER : Cycle 5, RW Stage (wrong!!!)
    * **CORRECT ANSWER** : End of Cycle 3, EX Stage

Forwarding



* If the correct value is already there in another stage, we can **forward** it.

Forwarding from MA to EX



* Fowarding in cycle 4 from instruction [1] to [2]

Different Forwarding Paths

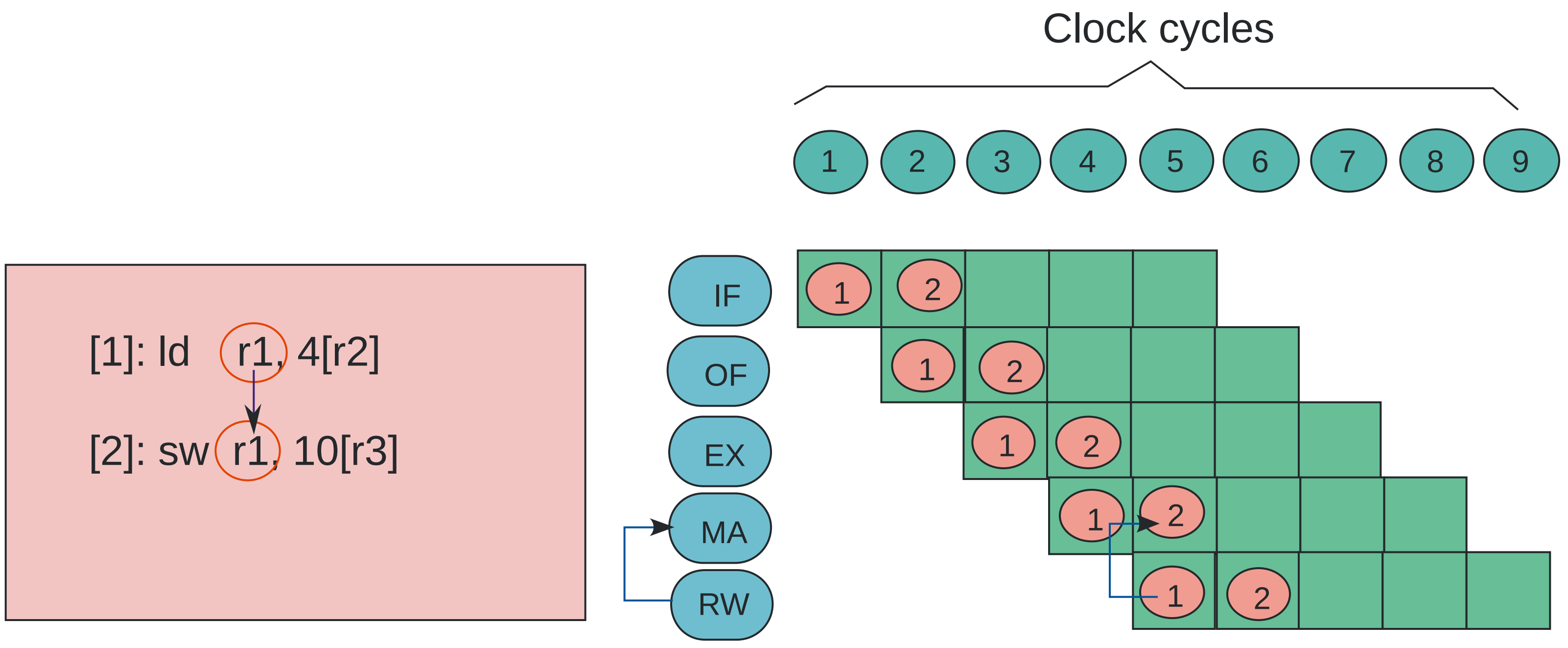
- We need to add a multitude of forwarding paths
- Rules for creating forwarding paths
  - Add a path from a later stage to an earlier stage
  - Try to add a forwarding path as late as possible. For, example, we avoid the EX $\longrightarrow$ OF forwarding path, since we have the MA $\longrightarrow$ EX forwarding path
  - The IF stage is not a part of any forwarding path.

Forwarding Path

- 3 Stage Paths
  - RW $\longrightarrow$ OF
- 2 Stage Paths
  - RW $\longrightarrow$ EX
  - MA $\longrightarrow$ OF (**X** Not Required)
- 1 Stage Paths
  - RW $\longrightarrow$ MA (load to store)
  - MA $\longrightarrow$ EX (ALU Instructions, load, store)
  - EX $\longrightarrow$ OF (**X** Not Required)

Mc Graw Hill Education

Forwarding Paths : RW → MA

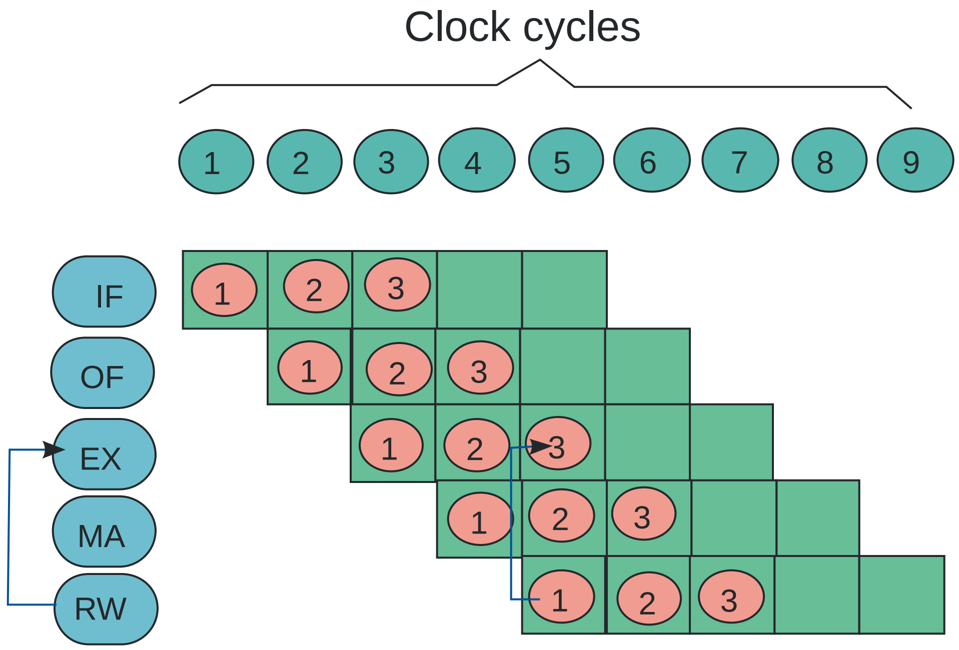


Forwarding Paths : RW → EX

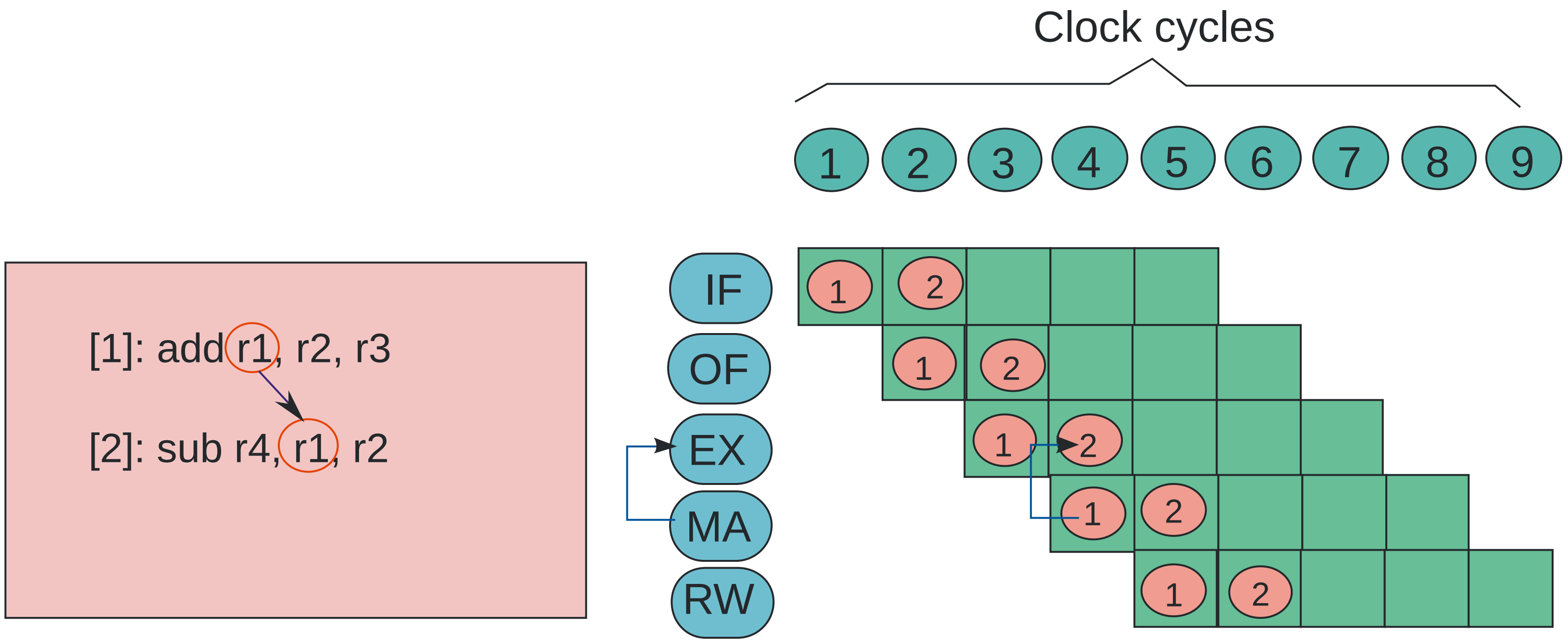
```
[1]: ld   r1, 4[r2]

[2]: sw  r8, 10[r3]

[3]: add r2, r1, r4
```

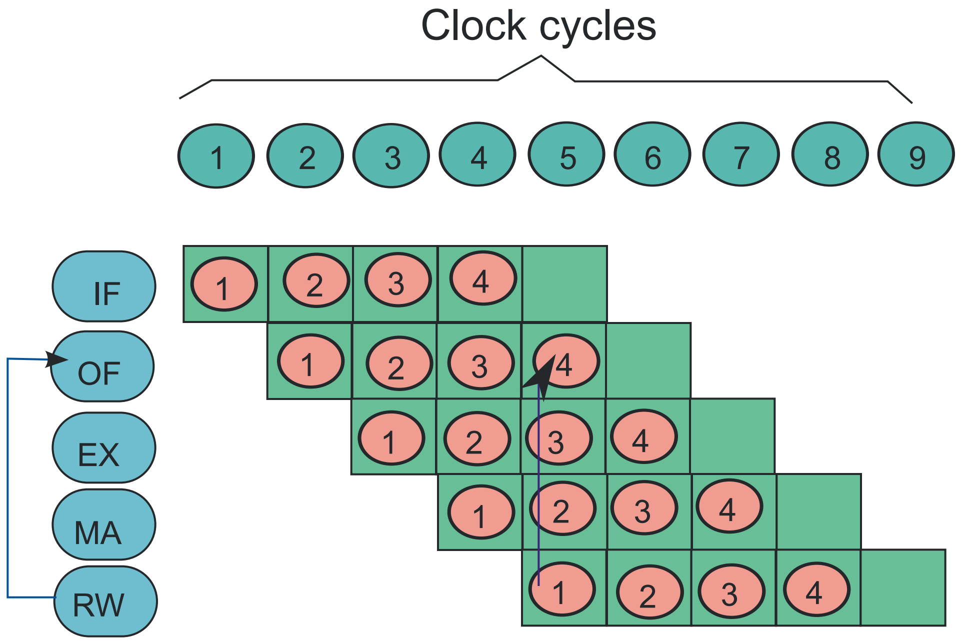


Forwarding Path : MA → EX



Forwarding Path : RW → OF

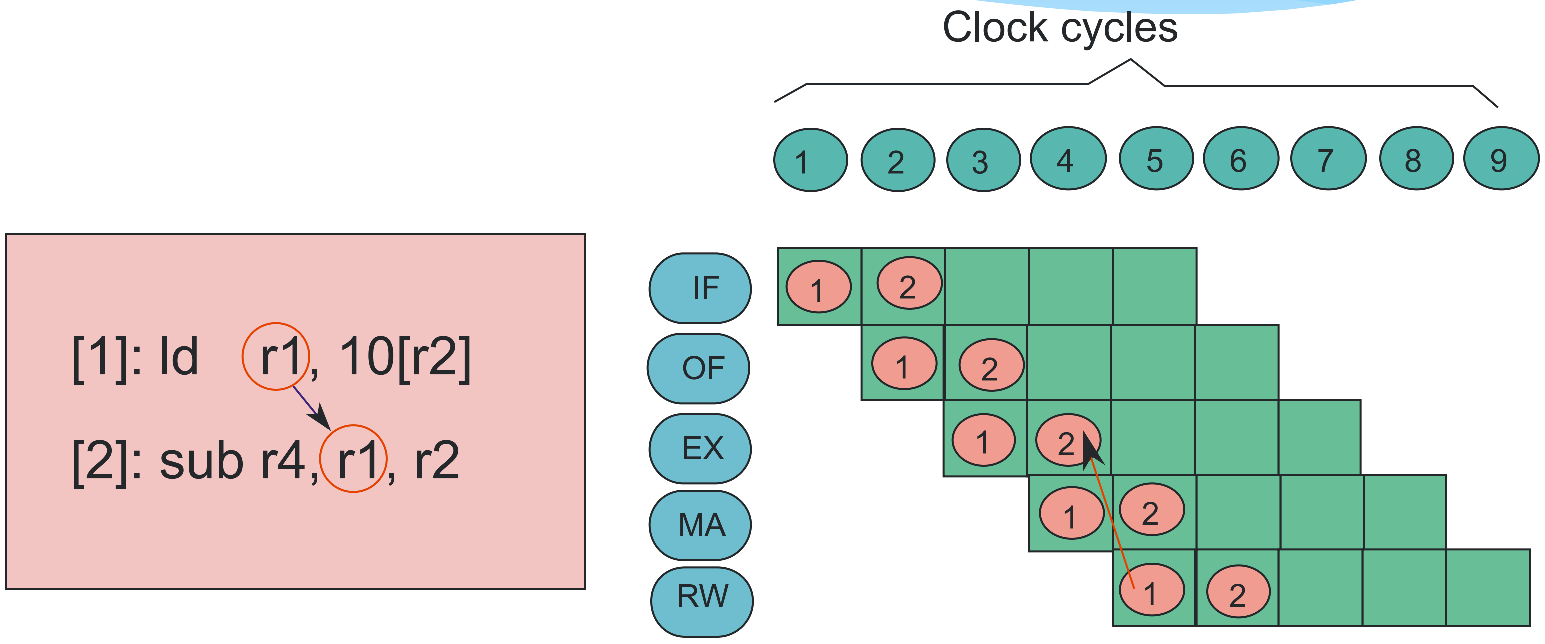
```
[1]: ld   r1, 4[r2]
[2]: sw   r4, 10[r3]
[3]: sw   r5, 10[r6]
[4]: sub r7, r1, r2
```



Data Hazards with Forwarding

* Forwarding has unfortunately not eliminated all data hazards
* We are left with one special case.
* Load-use hazard
  * The instruction immediately after a load instruction has a RAW dependence with it.

Load-Use Hazard

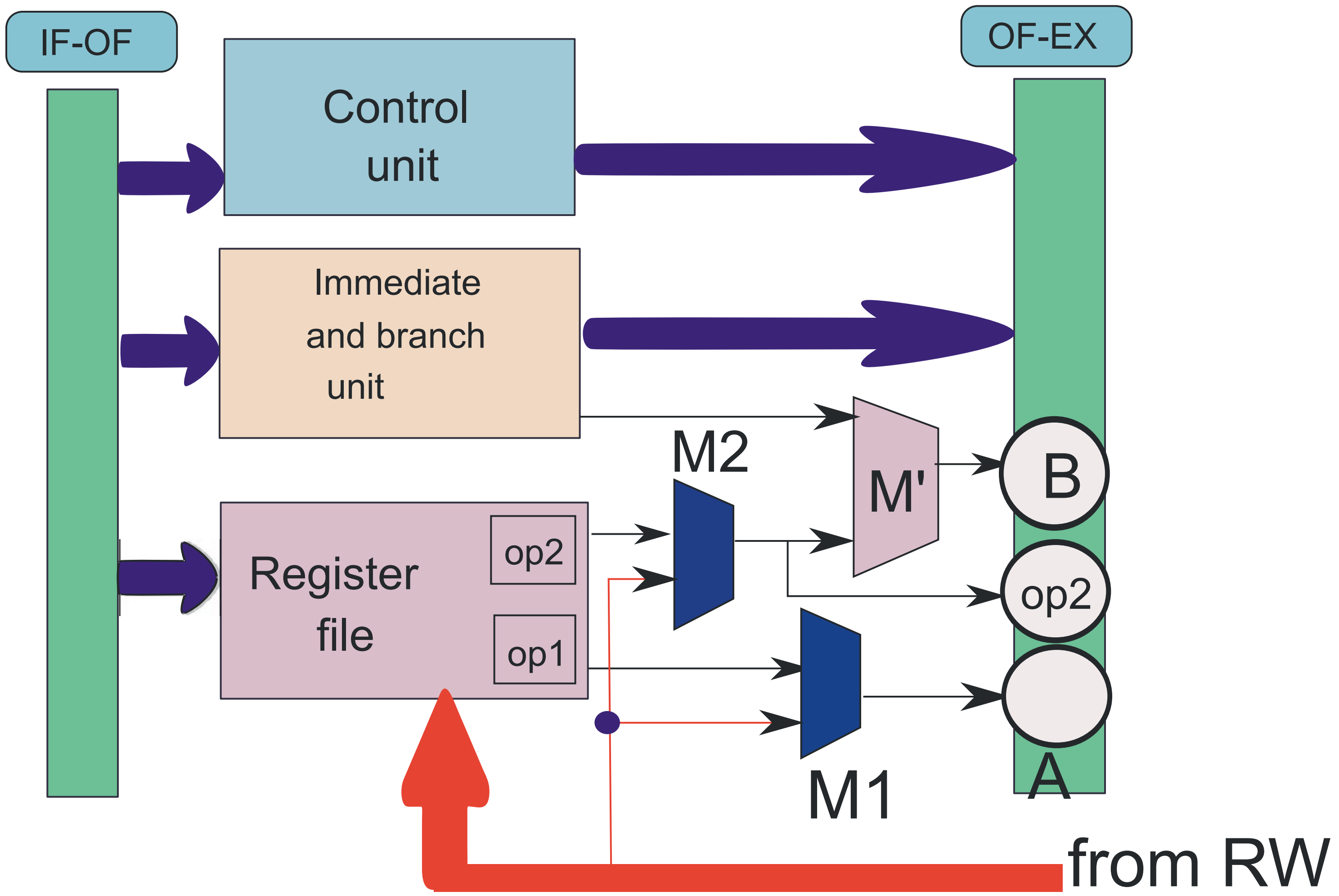


* Cannot forward (arrow goes backwards in time)
* Need to add a bubble (then use RW → EX forwarding)

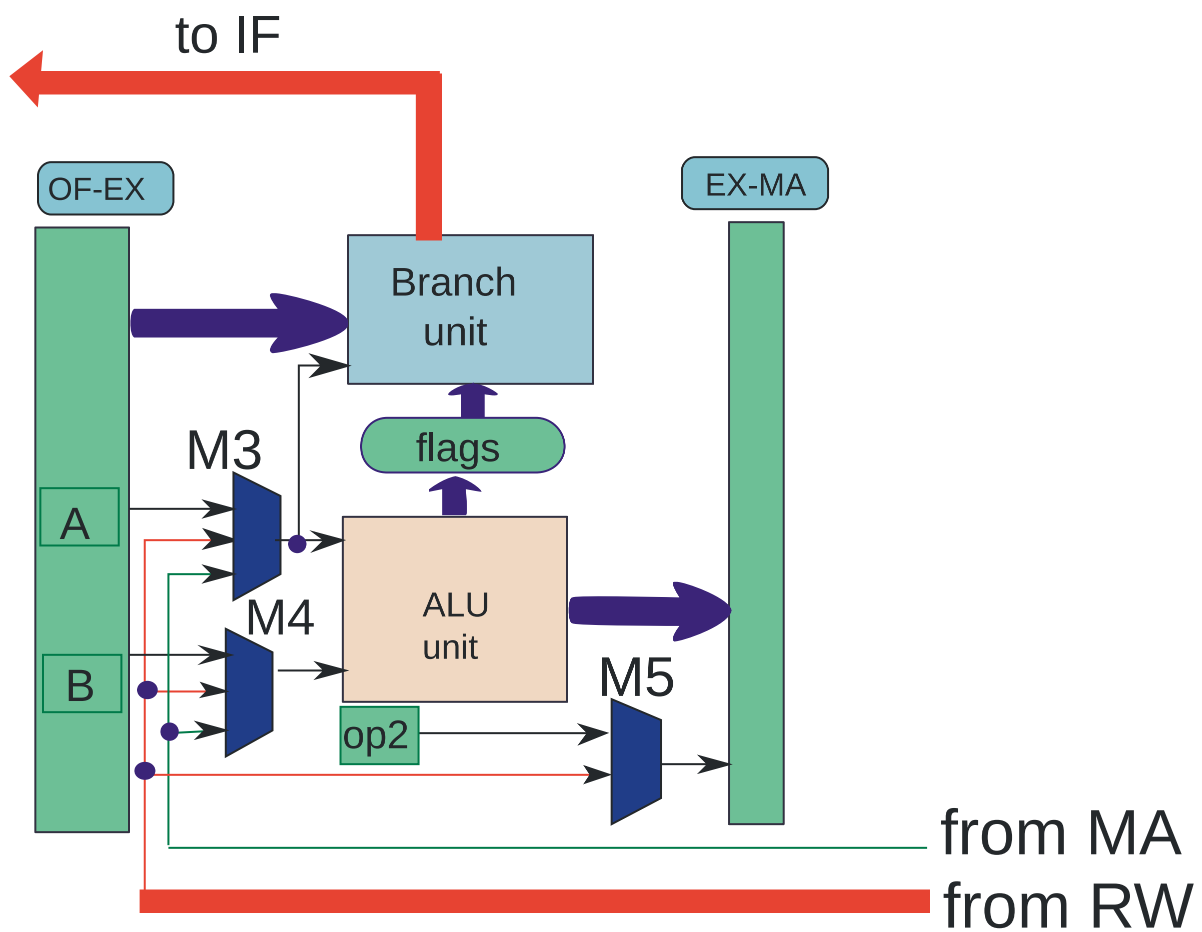
Implementation of Forwarding

* At every stage there is a choice of inputs for each functional unit
  * Use the default inputs from the previous stage
  * OR, use one of the forwarded inputs
  * Use a multiplexer to choose between the inputs
  * A dedicated forwarding unit generates the signals for the **forwarding multiplexers**

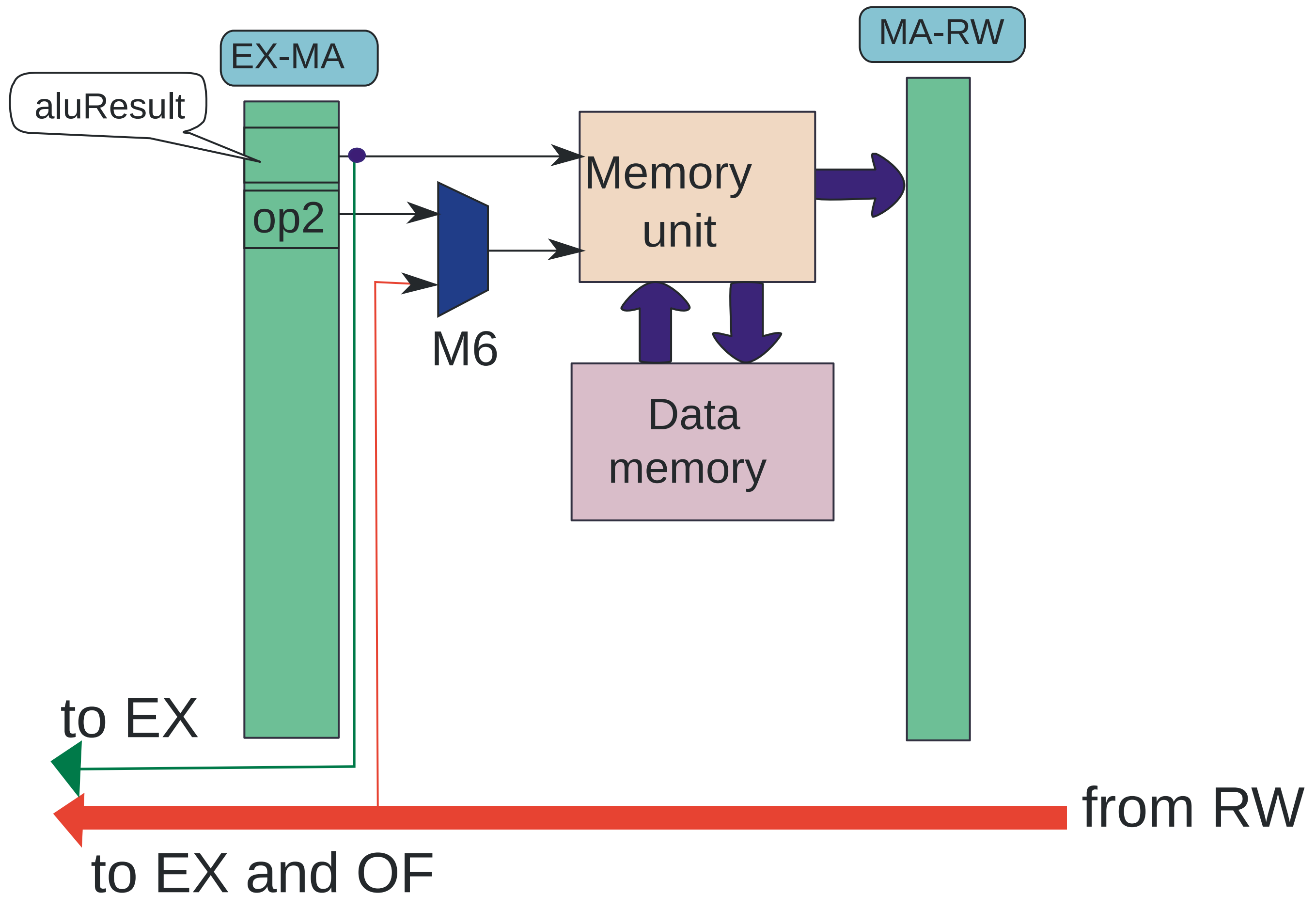
OF Stage with Forwarding



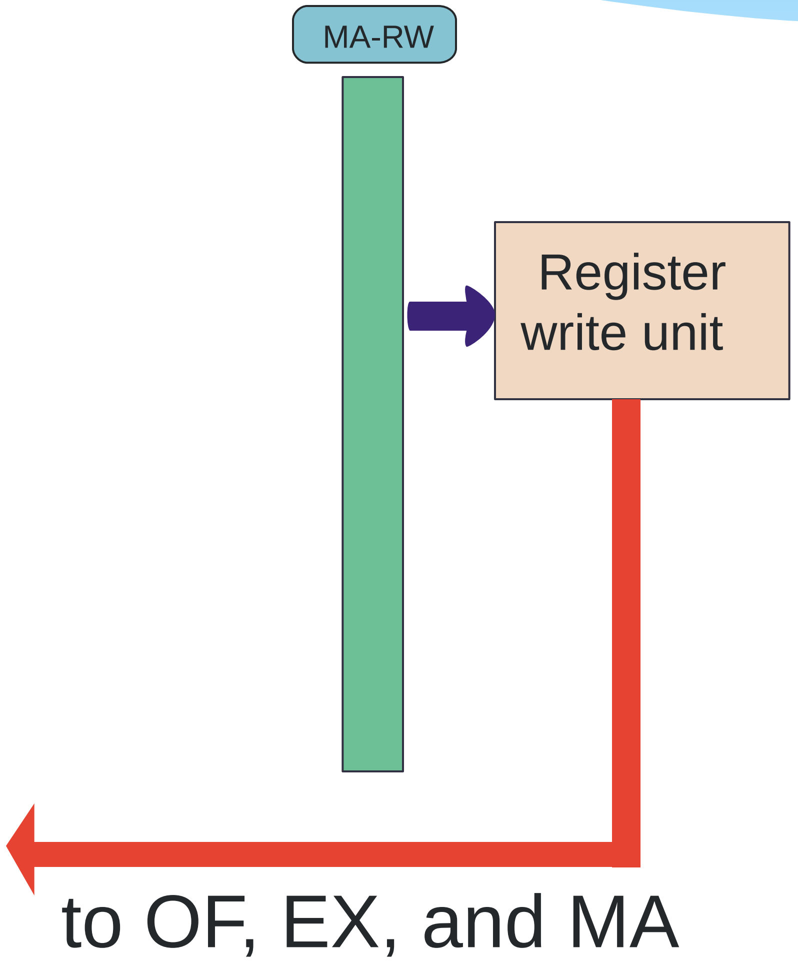
EX Stage with Forwarding



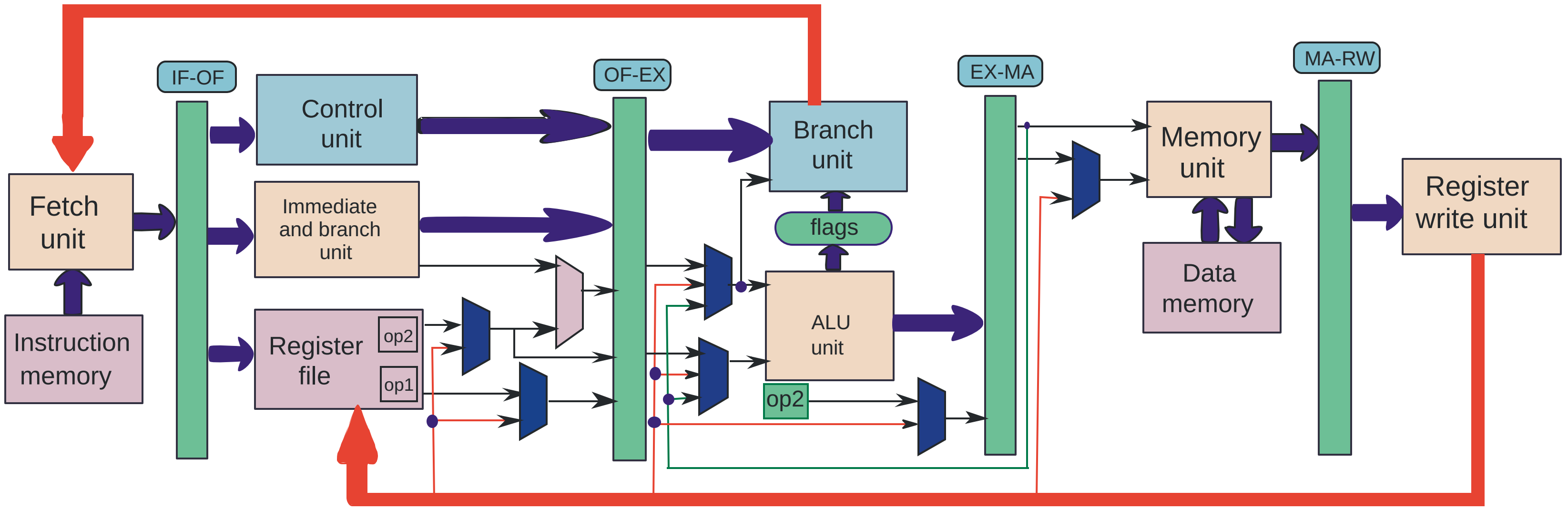
MA Stage with Forwarding



RW Stage with Forwarding



Data Path with Forwarding



Forwarding Conditions

- Determine if there is a conflict between two instructions in different stages
  - Find if there is a conflict for the first operand (rs1/ ra)
  - Find if there is a conflict for the second operand (rs2/rd)
- Always forward from the **latest** instruction (that is earlier than the current instruction)

```
Algorithm 6: Conflict on the first operand (rs1/ra)
Data: instructions, [A], and [B] (possible forwarding: [B] → [A])
Result: conflict exists on rs1/ra (true), no conflict (false)
if [A].opcode ∈ (nop,b,beq,bgt,call,not,mov) then
    /* Does not read from any register */
    return false
end
if [B].opcode ∈ (nop, cmp, st, b, beq, bgt, ret) then
    /* Does not write to any register */
    return false
end
/* Set the sources */
src1 ← [A].rs1
if [A].opcode = ret then
    src1 ← ra
end
/* Set the destination */
dest ← [B].rd
if [B].opcode = call then
    dest ← ra
end
/* Detect conflicts */
if src1 = dest then
    return true
end
return false
```

Algorithm 7: Conflict on the second operand (rs2/rd)
Data: instructions, [A], and [B] (possible forwarding: [B] → [A])
Result: conflict exists on second operand (rs2/rd) (**true**), no conflict (**false**)

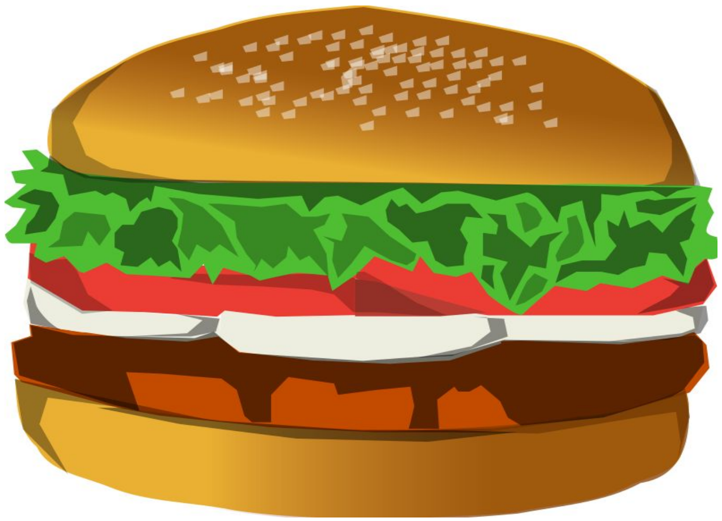
```
if [A].opcode ∈ (nop,b,beq,bgt,call) then
    /* Does not read from any register */
    return false
end
if [B].opcode ∈ (nop, cmp, st, b, beq, bgt, ret) then
    /* Does not write to any register */
    return false
end
/* Check the second operand to see if it is a register */
if [A].opcode ≠( st) then
    if [A].I = 1 then
        return false
    end
end
/* Set the sources */
src2 ← [A].rs2
if [A].opcode = st then
    src2 ← [A].rd
end
```

```
/* Set the destination */
dest ← [B].rd
if [B].opcode = call then
    dest ← ra
end
/* Detect conflicts */
if src2 = dest then
    return true
end
return false
```

Interlocks with Forwarding

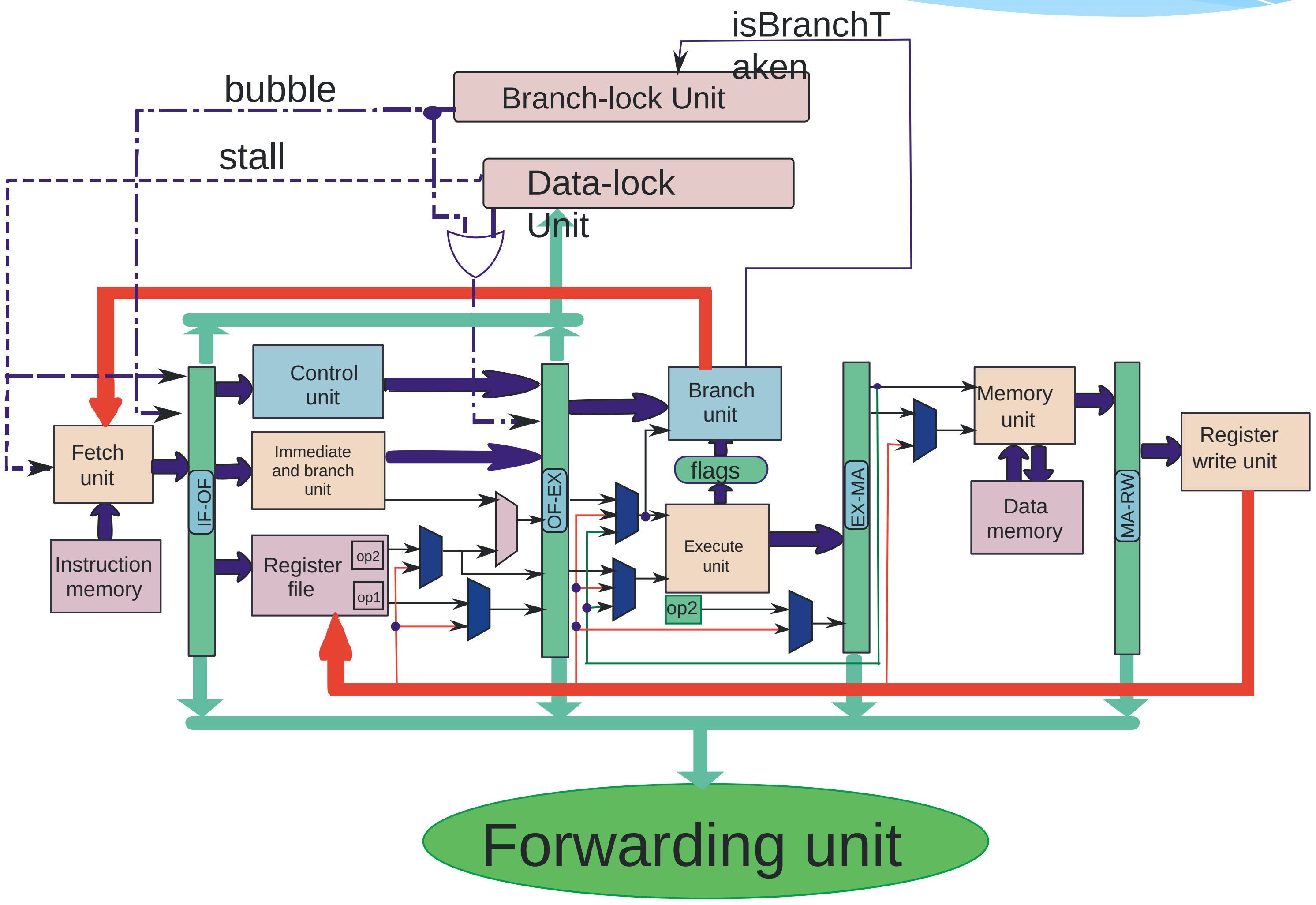
- Data-Lock
  - We need to only check for the load-use hazard
  - If the instruction in the EX stage is a load, and the instruction in the OF stage uses its loaded value, then stall for 1 cycle
- Branch-Lock
  - Remains the same as before.

The Curious Case of the *call* instruction



- The call instruction generates the value of ra in the RW stage
- Any instruction that uses the value of ra (such as ret) still works **correctly** !!!
- Prove it ....

Complete Data Path



Outline

- Overview of Pipelining
- A Pipelined Data Path
- Pipeline Hazards
- Pipeline with Interlocks
- Forwarding
- Performance Metrics
- Interrupts/ Exceptions

Mc Graw Hill Education

Measuring Performance

- What do we mean by the performance of a processor ?
  - **ANSWER** : Almost nothing
- What should we ask instead ?
  - What is the performance with respect to a given program or a set of programs ?
  - Performance is inversely proportional to the time it takes to execute a **program**

Computing the Time a Program Takes

$$
\begin{aligned}
\tau &= \#seconds \\
&= \frac{\#seconds}{\#cycles} * \frac{\#cycles}{\#instructions} * (\#instructions) \\
&= \underbrace{\frac{\#seconds}{\#cycles}}_{1/f} + \underbrace{\frac{\#cycles}{\#instructions}}_{CPI} * (\#instructions) \\
&= \frac{CPI * \#insts}{f}
\end{aligned}
$$

* CPI ⟶ Cycles per instruction
* f ⟶ frequency (cycles per second)

The Performance Equation

$$P \propto \frac{IPC * f}{\#insts}$$

- IPC $\longrightarrow$ 1/CPI (Instructions per Cycle)
- What are the units of performance ?
  - **ANSWER** : arbitrary

Number of Instructions (#insts)

Static Instruction: The binary or executable of a program, contains a list of ***static instructions.***

Dynamic Instruction: A **dynamic instruction** is a running instance of a static instruction, which is created by the processor when an instruction enters the pipeline.

- Note that these are dynamic instructions
  - **NOT** static instructions
- A smart compiler can reduce the number of executed instructions

Number of Instructions(#insts) – 2

* Dead code removal
  * Often programmers write code that does not determine the final output
  * This code is redundant
  * It can be identified and removed by the **compiler**
* Function inlining
  * Very small functions have a lot of overhead → call, ret instructions, register spilling, and restoring
  * Paste the code of the callee in the code of the caller (known as **inlining**)

Computing the CPI

- CPI for a single cycle processor = 1
- CPI for an ideal pipeline (no hazards)
    - Assume we have n instructions, and k stages

Computing the CPI

- CPI for a single cycle processor = 1
- CPI for an ideal pipeline(no hazards)
  - Assume we have *n* instructions, and *k* stages
  - The first instruction enters the pipeline in cycle 1
  - It leaves the pipeline in cycle k
  - The rest of the **(n-1) instructions** leave in the next (n-1) consecutive cycles

$$CPI = \frac{n + k - 1}{n}$$

Computing the Maximum Frequency

- Let the maximum amount of time that it takes to execute any instruction be :
  - t_{max} (also known as algorithmic work)
- Minimum clock cycle time of a single cycle pipeline $\longrightarrow t_{max}$
- In the case of a pipeline, let us assume that all the pipeline stages are **balanced**
- Time per stage $\longrightarrow t_{max} / k$

Maximum Frequency - II

* Let the **latch delay** be **l**
* We thus have :

$$t_{stage} = \frac{t_{max}}{k} + l$$

$$\frac{1}{f} = \frac{t_{max}}{k} + l$$

The minimum cycle time (1/f) is equal to t_{stage} . Let us thus, assume that our cycle time is as low as possible.

Performance of an Ideal Pipeline

* Let us assume that the number of instructions are a **constant**

$$
\begin{aligned}
P &= \frac{f}{CPI} \\
&= \frac{\frac{1}{\frac{t_{max}}{k}+l}}{\frac{n+k-1}{n}} \\
&= \frac{n}{\left(\frac{t_{max}}{k}+l\right)*(n+k-1)} \\
&= \frac{n}{\frac{(n-1)t_{max}}{k}+(t_{max}+\ln-l)+lk}
\end{aligned}
$$

Optimal Number of Pipeline Stages

$$\frac{\partial\left(\frac{(n-1)t_{max}}{k} + (t_{max} + \ln - l) + lk\right)}{\partial k} = 0$$

$$\Rightarrow -\frac{(n-1)t_{max}}{k^2} + l = 0$$

$$\Rightarrow k = \sqrt{\frac{(n-1)t_{max}}{l}}$$

- k is inversely proportional to $\sqrt{l}$
- k is proportional to $\sqrt{t_{max}}$

Implications

* As we increase the latch delay, we should have less pipeline stages
  * We need to minimise the time wasted in accessing latches
* As we increase the amount of **algorithmic work**, we require more pipeline stages for ideal performance
  * More pipeline stages help distribute the work better, and increase the overlap across instructions

A Non-Ideal Pipeline

* Our ideal CPI (CPI$_{ideal}$ = 1) is 1
* However, in reality, we have stalls

$$CPI = CPI_{ideal} + stall\_rate * stall\_penalty$$

* Let us assume that the stall rate is a function of the program, and its nature of dependences

Non-Ideal Pipeline - II

* Let us assume that the stall penalty is proportional to the number of pipeline stages
* Both these assumptions are strictly not correct. They are being used to make a coarse grained mathematical model.
* CPI = (n+k-1)/n + rck
  * r $\longrightarrow$ stall rate, c $\longrightarrow$ constant of proportionality

Mathematical Model

$$P = \frac{f}{CPI}$$

$$= \frac{\frac{1}{\frac{t_{max}}{k} + l}}{\frac{n+k-1}{n} + rck}$$

$$= \frac{n}{\frac{(n-1)t_{max}}{k} + (rcnt_{max} + \ t_{max} + \ln - l) + lk(1 + rcn)}$$

Mathematical Model - II

$$\frac{\partial\left(\frac{(n-1)t_{max}}{k} + (rcnt_{max} + \ t_{max} + \ln - l) + lk(1+rcn)\right)}{\partial k} = 0$$

$$\Rightarrow - \frac{(n-1)t_{max}}{k^2} + l(1+rcn) = 0$$

$$\Rightarrow k \ = \sqrt{\frac{(n-1)t_{max}}{l(1+rcn)}} \approx \sqrt{\frac{t_{max}}{lrc}} \quad (as\, n \ \longrightarrow \ \infty)$$

Implications

* For programs with a lot of dependences (high value of r) ⟶ Use less pipeline stages
* For a pipeline with forwarding ⟶ c is smaller (than a **pipeline** that just has interlocks)
    * It requires a larger number of pipeline stages for optimal performance

Implications

- The optimal number of pipeline stages is directly proportional to $\sqrt{(t_{max} / l)}$
  - This ratio is not significantly changing across technologies.
  - This explains why the number of pipline stages has remained more or less constant for the last 5-10 years

Example

Example Consider two programs that have the following characteristics.

Program 1		Program 2	
Instruction Type	Fraction	Instruction Type	Fraction
loads	0.4	loads	0.3
Branches	0.2	Branches	0.1
ratio(taken branches)	0.5	ratio(taken branches)	0.4

The ideal CPI is 1 for both the programs. Let 50% of the load instructions suffer from a load use hazard. Assume that the frequency of P_1 is 1, and the frequency of P_2 is 1.5. Here, the units of the frequency are not relevant. Compare the performance of P_1 and P_2.

Example

Answer:

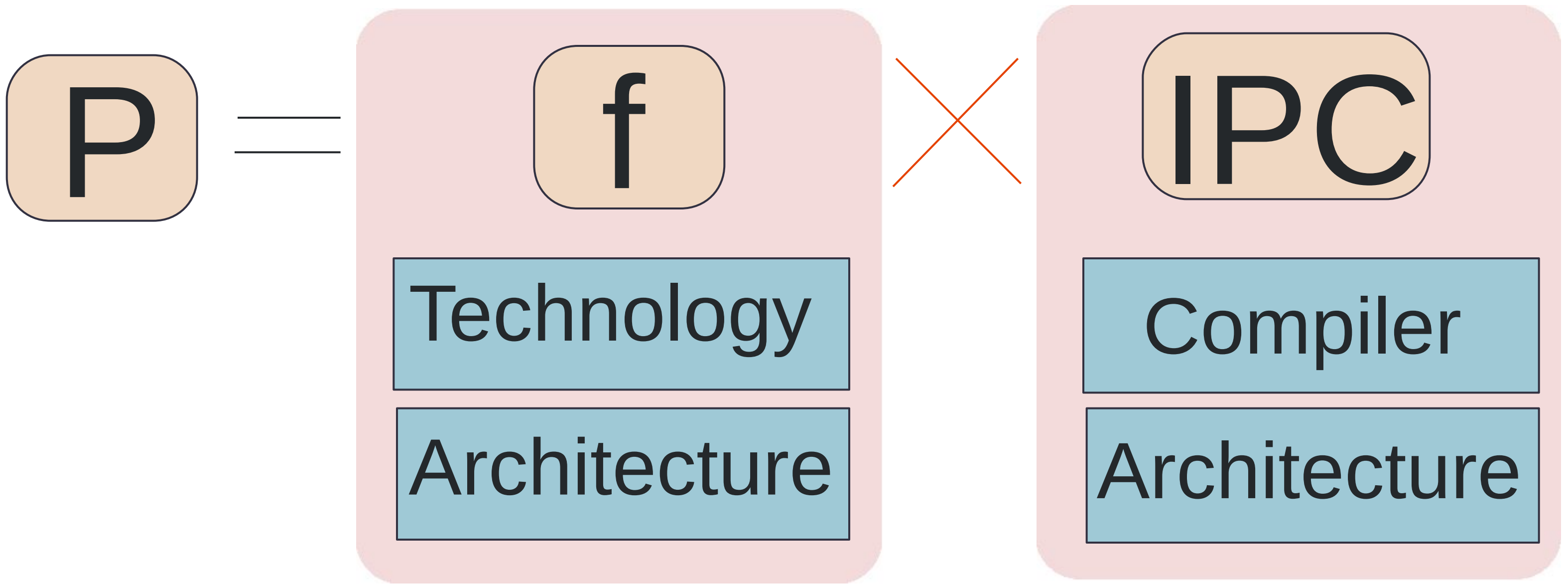
$$\begin{aligned} CPI_{new} =& CPI_{ideal} + 0.5 \times (ratio(loads)) \times 1 \\ & + ratio(branches) \times ratio(taken\,branches) \times 2 \end{aligned} \tag{9.13}$$

We thus have:

$$CPI_{P_1} = 1 + 0.5 \times 0.4 + 0.2 \times 0.5 \times 2 = 1 + 0.2 + 0.2 = 1.4$$
$$CPI_{P_2} = 1 + 0.5 \times 0.3 + 0.1 \times 0.4 \times 2 = 1 + 0.15 + 0.08 = 1.23$$

The performance of P_1 can be expressed as f/CPI = 1 / 1.4 = 0.71 (arbitrary units). Similarly, the performance of P_2 is equal to f/CPI = 1.5/1.23 = 1.22 (arbitrary units). Hence, P_2 is faster than P_1. We shall often use the term, arbitrary units, a.u., when the choice of units is irrelevant.

Performance, Architecture, Compiler



Outline

* Overview of Pipelining
* A Pipelined Data Path
* Pipeline Hazards
* Pipeline with Interlocks
* Forwarding
* Performance Metrics
* Interrupts/ Exceptions

What happens when you press a key ?

* The keyboard logs the key press
  * Converts the key to ASCII or Unicode
  * Sends the code to the processor
  * The processor thus receives an **interrupt**
  * It suspends the current program
  * Jumps to the interrupt handler.
  * The interrupt handler draws the shape associated with the key
  * The processor returns to execute the original program

Exceptions

- Exceptions are generated when
  - A program accesses an illegal address
  - We try to divide 5/0
  - We issue an invalid instruction
  - …
- Exception are treated the same way as interrupts
  - Jump to the exception handler
  - Come back and start executing programs

Precise Exceptions

- Informal definition
  - We need to return to the original program at exactly the same point, at which we had left it
  - The execution of the interrupt handler should not disrupt the execution of the original program in any way. The outcome of the original program should be independent of the interrupt (unless the program caused an exception).

Precise Exceptions - II

* Formal Definition
  * Let us number the dynamic instructions in a program : $I_1 \ldots I_n$
  * Let us assume that an instruction completes after it either updates memory, writes to registers, or reaches the MA stage (cmp, b, beq, bgt, ret)
  * Let the last program instruction that completes before the first instruction in the interrupt handler completes, be I_k
  * Let all the program instructions that complete before the first instruction in the interrupt handler completes, be **C**

Precise Exceptions - III

$$I_j \in C \Leftrightarrow (j \leq k)$$

- We need to ensure this condition.
- Also no instruction of the form $I_{k'}$ (k' > k) should complete before all the instructions in the interrupt handler complete
- After returning, we can seamlessly execute I_k(same instruction)or I_{k+1} (next instruction)

Marking Instructions

* When an interrupt arrives, let us mark the instruction in the MA stage
* Otherwise, if there is an exception
  * We mark the instruction, as soon as it encounters a fault/exception
* Once, an insruction is marked, we have two kinds of instructions in the pipeline
  * Instructions before/after the **marked** instruction

Implementing Precise Exceptions

* Wait till a marked instruction reaches the end of the pipeline
  * Convert all the instructions after the marked instruction to bubbles
* Ensures that both the conditions of a precise exception are met.
* Once the marked instruction reaches the end of the pipeline, the exception unit loads the pc of the interrupt handler

Saving/Restoring Program State

- Program State
  - PC
  - Registers
  - Flags
  - Memory
- Memory → Assume that there is no overlap of memory regions, unless explicitly intended

oldPC Register

- Let us add a npc field in the instruction packet
  - For taken branches it is equal to the branch target
  - For all other instructions it is equal to (pc + 4)
- We populate the npc field in the EX stage
- Depending on the type of the exception, we might want to return to pc or npc
  - The exception unit sets the **oldPC** register to the right return address

Spilling/ Restoring Registers

* In the case of functions, we stored registers on the stack
  * In this case, the interrupt handler has a separate stack.
  * We cannot overwrite the stack pointer (we will lose its previous value)
* Solution :
  * Use an additional register, **oldSP**, to save the stack pointer of the program.
  * Load the new stack pointer, and spill all the registers
  * Save oldPC

The Strange Case of the Flags

- **Naive solution** :
  - Do not allow any instruction after the marked instruction to update the flags register
  - We detect an exception typically towards the middle or end of a cycle
  - By that time, the instruction might have already updated the flags register (at least the master latch)

Solution

- Add a flags field to the instruction packet
- Every instruction saves the updated value of the flags register to the flags field in its instruction packet.
- The exception unit saves the flags field of the marked instruction to the oldFlags register
- We store the oldFlags register to a location in the interrupt handler's stack

Privileged Instructions

- We add the following new registers to the ISA
  - special registers → flags, oldFlags, oldPC, oldSP
  - The flags register is now accessible to the ISA
- Add the z series of priveleged instructions
- movz instruction
  - Transfers values between regular registers and special registers

Priveleged Instructions - II

* retz
  * pc ⟵ oldPC
* The z series instructions can only be used by the interrupt handler, and the operating system
* We use a CPL bit (current privelege level) bit for setting permissions
  * User programs (CPL = 1)
  * Interrupt handlers, kernel programs, (CPL = 0)

Implementing *movz* and *retz*

- Define two new opcodes
  - These opcodes are usable only when CPL = 0
  - Augment the OF and RW stages to use the special registers.
  - movz, and retz see a different view of registers.

Register	Encoding
r0	0000
oldPC	0001
oldSP	0010
flags	0011
oldFlags	0100
sp	1110

Assembly Code for Spilling Registers

```
/* save the stack pointer */
movz oldSP, sp
mov sp, 0x FF FC

/* spill all the registers other
than sp*/
st r0, -4[sp]
st r1, -8[sp]
st r2, -12[sp]
st r3, -16[sp]
st r4, -20[sp]
st r5, -24[sp]
st r6, -28[sp]
st r7, -32[sp]
st r8, -36[sp]
st r9, -40[sp]
st r10, -44[sp]
st r11, -48[sp]
st r12, -52[sp]
```

Spilling Registers - II

```
st r13, -56[sp]
st r15, -60[sp]

/* save the stack pointer */
movz r0, oldSP
st r0, -64[sp]

/* save the flags register */
movz r0, oldFlags
st r0, -68[sp]

/* save the oldPC */
movz r0, oldPC
st r0, -72[sp]

/* update the stack pointer */
sub sp, sp, 72
/* code of the interrupt handler */
....
....
....
```

Restoring Registers

```
/* update the stack pointer */
add sp, sp, 72
/* restore the oldPC register */
ld r0, -72[sp]
movz oldPC, r0

/* restore the flags register */
ld r0, -68[sp]
movz flags, r0


/* restore all the registers other than sp*/
ld r0, -4[sp]
ld r1, -8[sp]
ld r2, -12[sp]
ld r3, -16[sp]
ld r4, -20[sp]
ld r5, -24[sp]
ld r6, -28[sp]
ld r7, -32[sp]
ld r8, -36[sp]
```

Restoring Registers – II

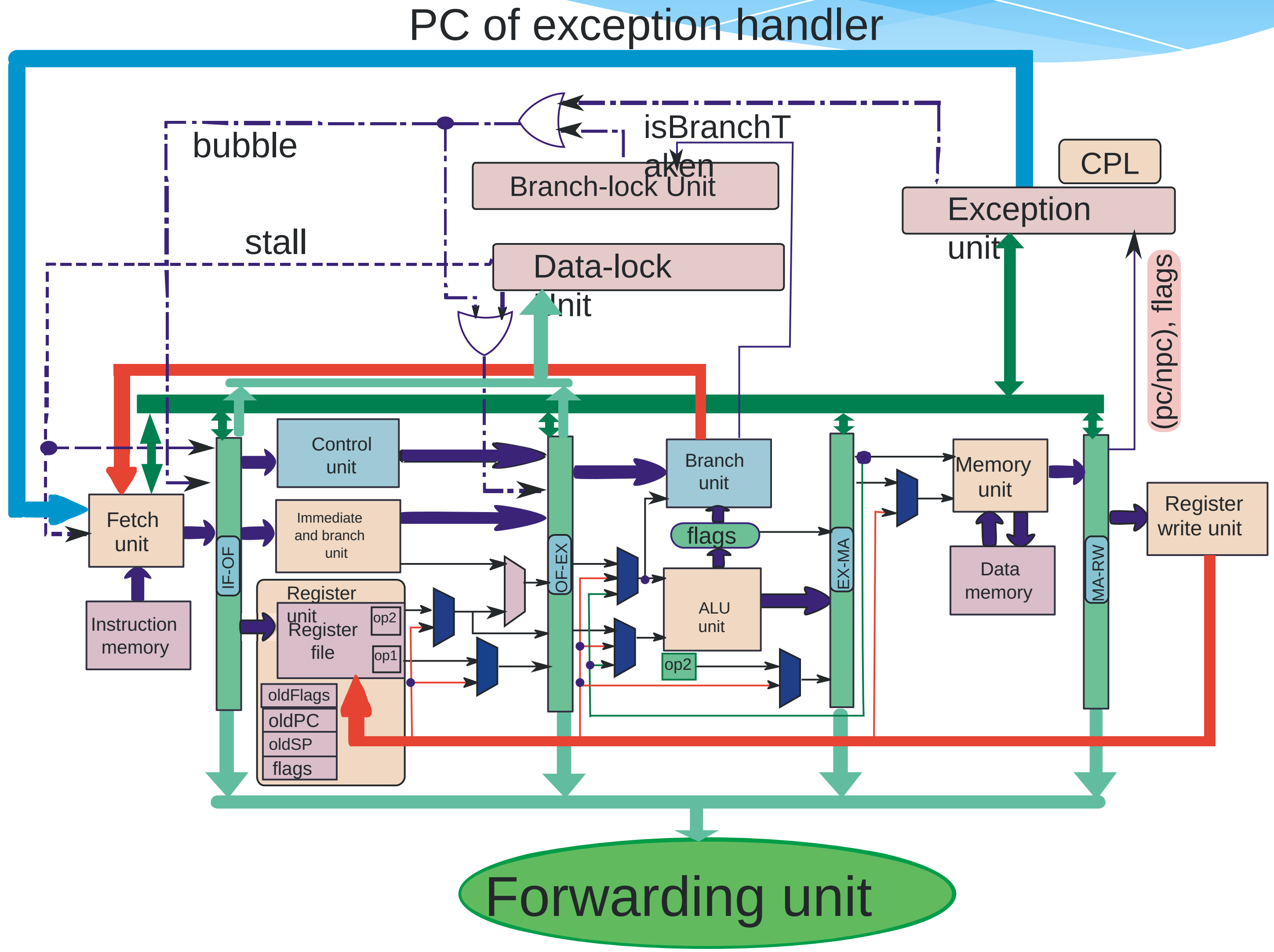
```
ld r9, -40[sp]
ld r10, -44[sp]
ld r11, -48[sp]
ld r12, -52[sp]
ld r13, -56[sp]
ld r15, -60[sp]

/* restore the stack pointer */

ld sp, -64[sp]

/* return to the program */

retz
```


PC of exception handler
bubble
isBranchTaken
Branch-lock Unit
CPL
Exception unit
stall
Data-lock Unit
(pc/npc), flags
Control unit
Branch unit
Memory unit
Fetch unit
Immediate and branch unit
flags
Register write unit
Instruction memory
IF-OF
OF-EX
EX-MA
MA-RW
Register unit
Register file
op2
op1
ALU unit
op2
Data memory
oldFlags
oldPC
oldSP
flags
Forwarding unit

THE END