# Advanced Computer Architecture

## ILP Boost: Dynamic Scheduling

Debadatta Mishra, CSE, IITK

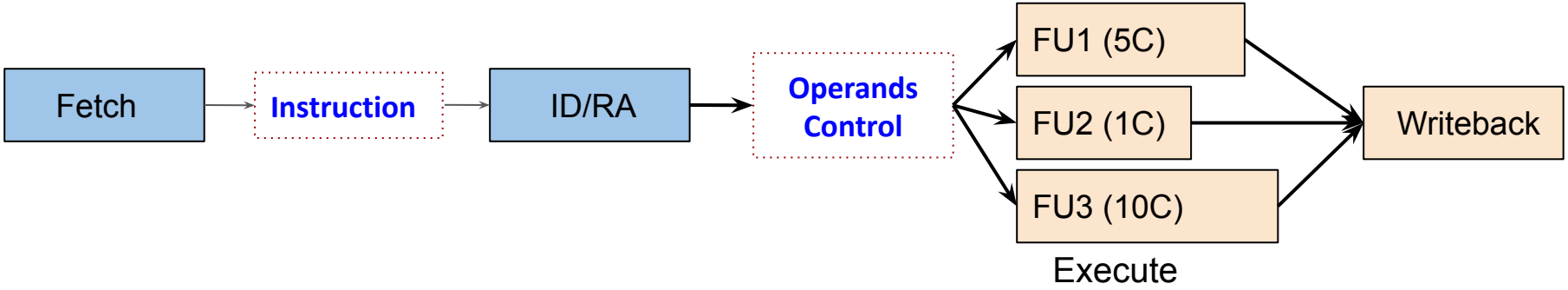
# Recap: Pipeline is good, but …

- Implications of pipelining a single cycle implementation into multiple stages
  - Splitting stages for non-overlapped execution, cost vs. performance tradeoff
  - **Balancing stages is critical – slowest stage determine clock cycle**
  - Additional overhead to maintain pipeline registers
- Even with perfectly balanced pipelines, pipeline can perform poorly because of
  - Data dependence
  - Disruption of sequential execution flow
    - Program structure (`if-else`, `loops`, `switch-case`, `procedure call/ret`)
    - Exceptions, system calls

How to balance the pipeline stages?

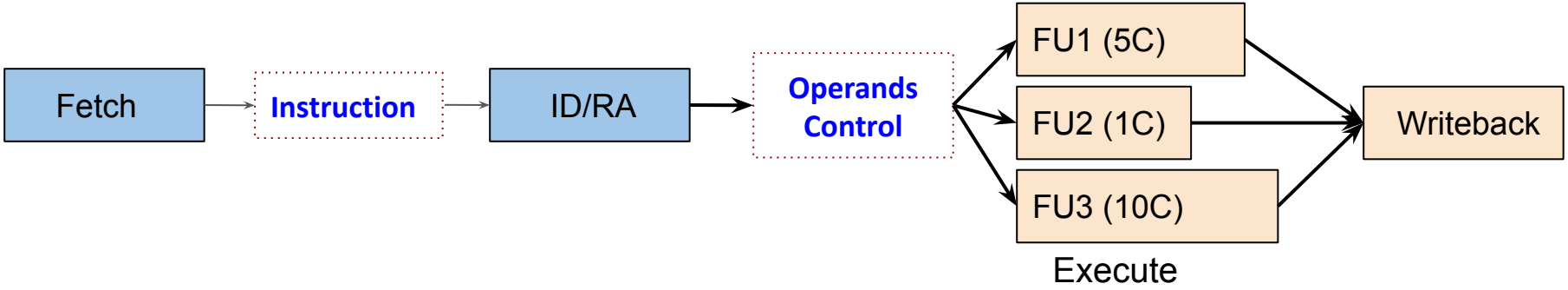
Is it possible to statically determine the stages?

# Deciding pipeline depth: Multi-cycle Operations



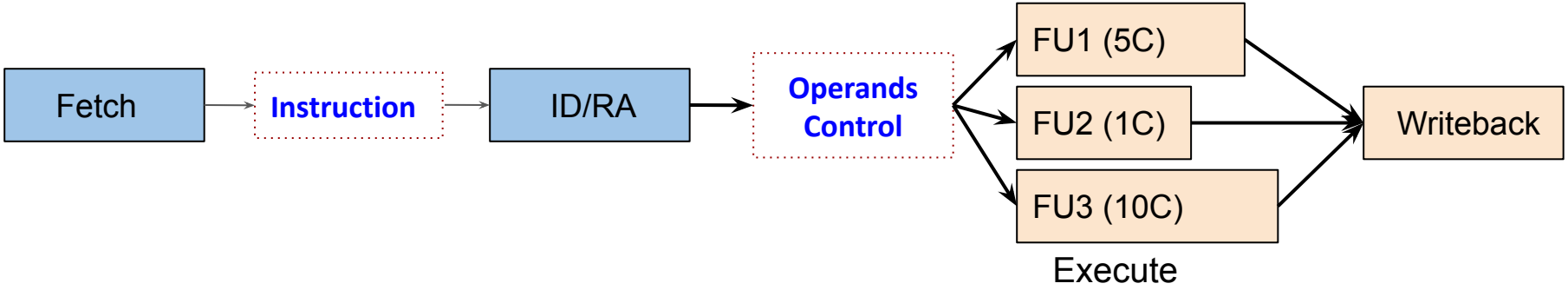

- Even assuming LD/ST takes one cycle, how would the pipeline perform?
- What are the optimization strategies?

# Deciding pipeline depth: Multi-cycle Operations



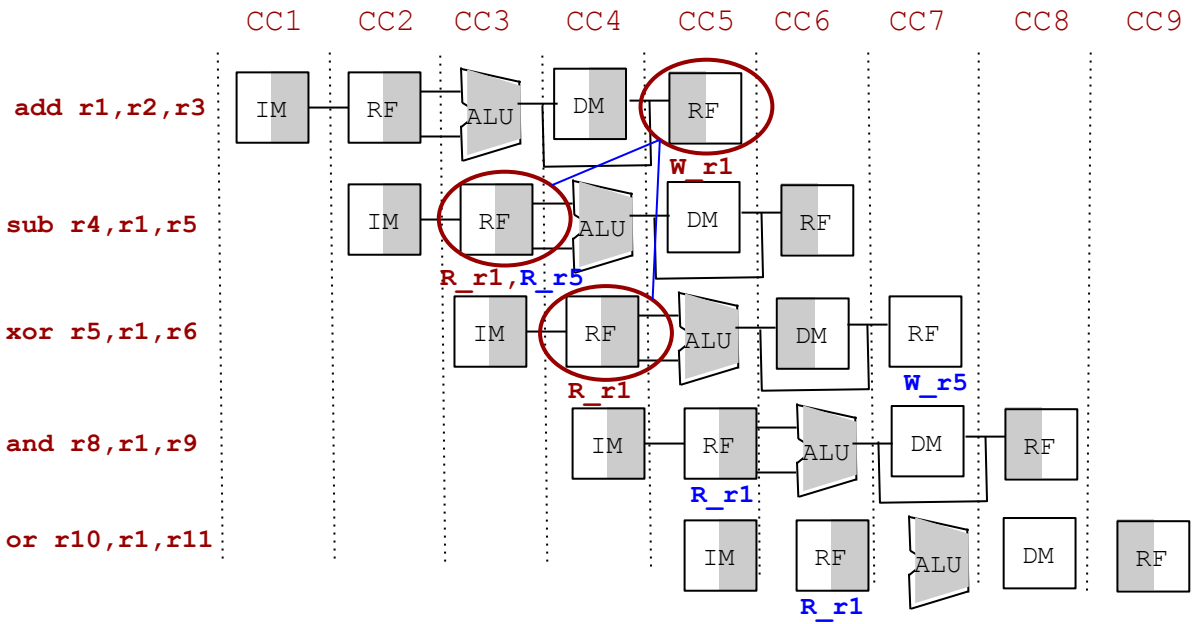

- Even assuming LD/ST takes one cycle, how would the pipeline perform?
  - The pipeline stalls with instruction mix requiring different FUs
  - Pipeline with 'Ex' split into 10 cycles⟶ ?
  - Increasing the clock to accommodate slowest operation ⟶ ?

# Deciding pipeline depth: Multi-cycle Operations



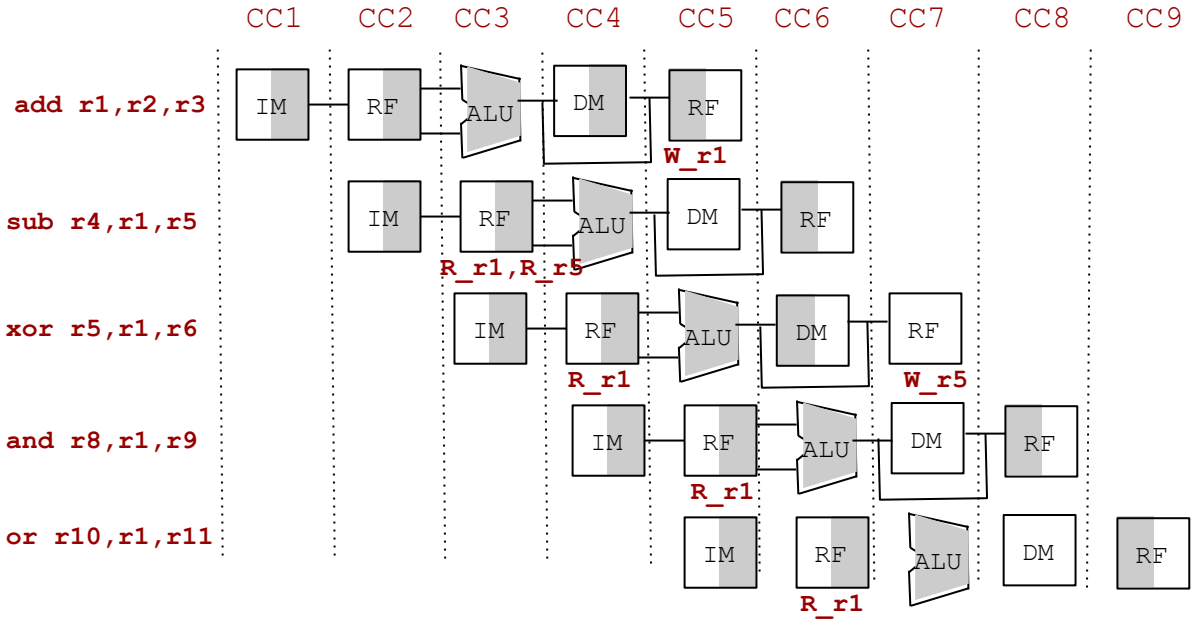

- Even assuming LD/ST takes one cycle, how would the pipeline perform?
  - The pipeline stalls with instruction mix requiring different FUs
  - Pipeline with 'Ex' split into 10 cycles⟶ reduced IPC
  - Increasing the clock to accommodate slowest operation ⟶ slow processor
- Batching instructions (i.e., reordering) requiring same FUs helps assuming
  - FUs are pipelined
  - Dependencies remain a concern

# Data hazard (recap)



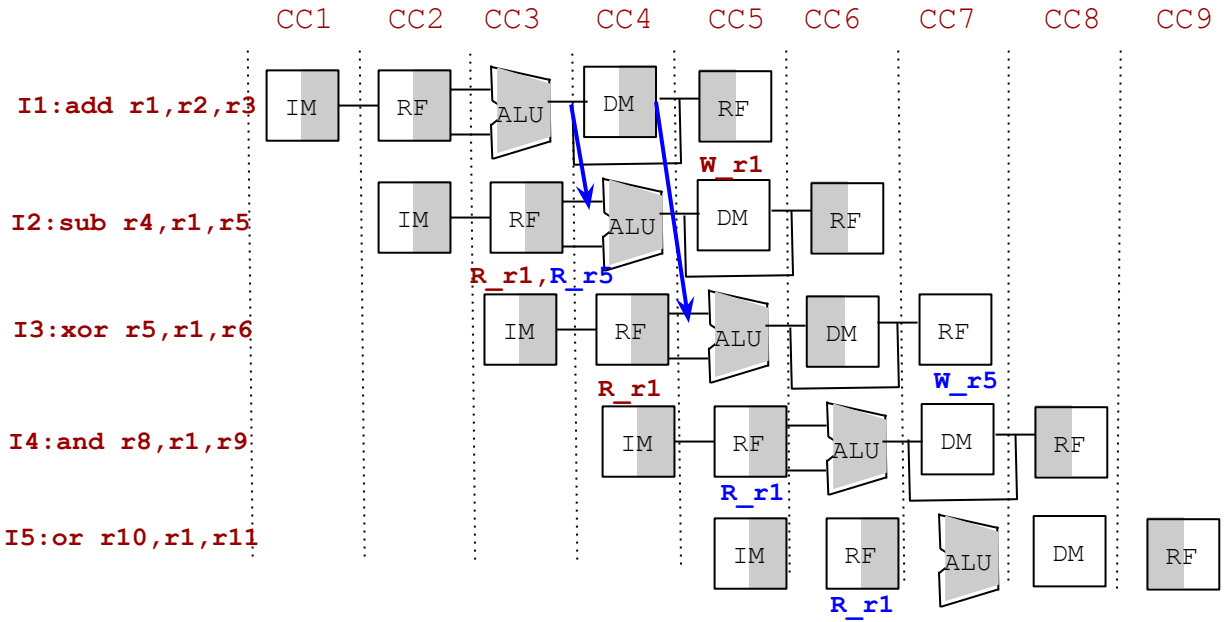

- Register **r1** is written in CC5 (first half of the clock) by **add**
- **r1** is read by **sub,xor,and, or** in the second half of CC3, CC4, CC5 and CC6, respectively
- Which instructions can not proceed? For how long?
- **sub** for 2 cycles and **xor** for 1 cycle

# Data hazard (recap)



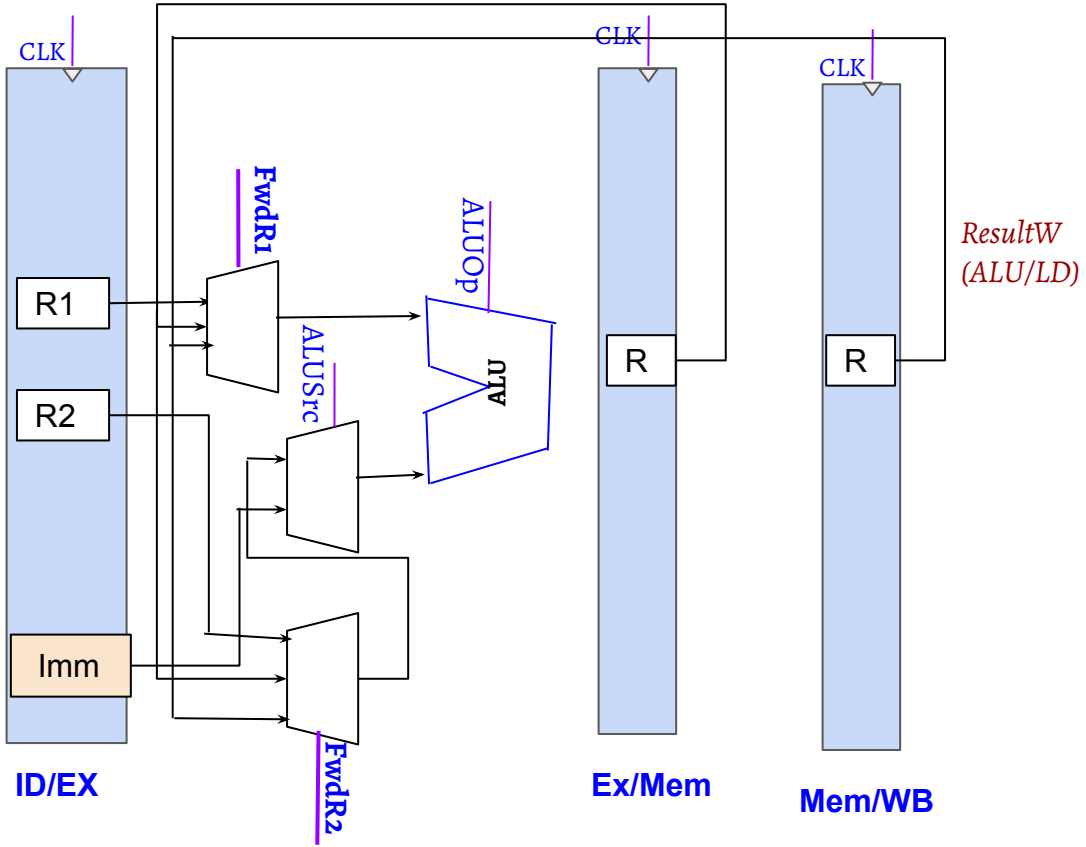

- Register **`r1`** is written in CC5 (first half of the clock) by **`add`**
- **`r1`** is read by **`sub`**,**`xor`**,**`and`**, **`or`** in the second half of CC3, CC4, CC5 and CC6, respectively
- Which instructions can not proceed?

# Data hazard: Bypass network



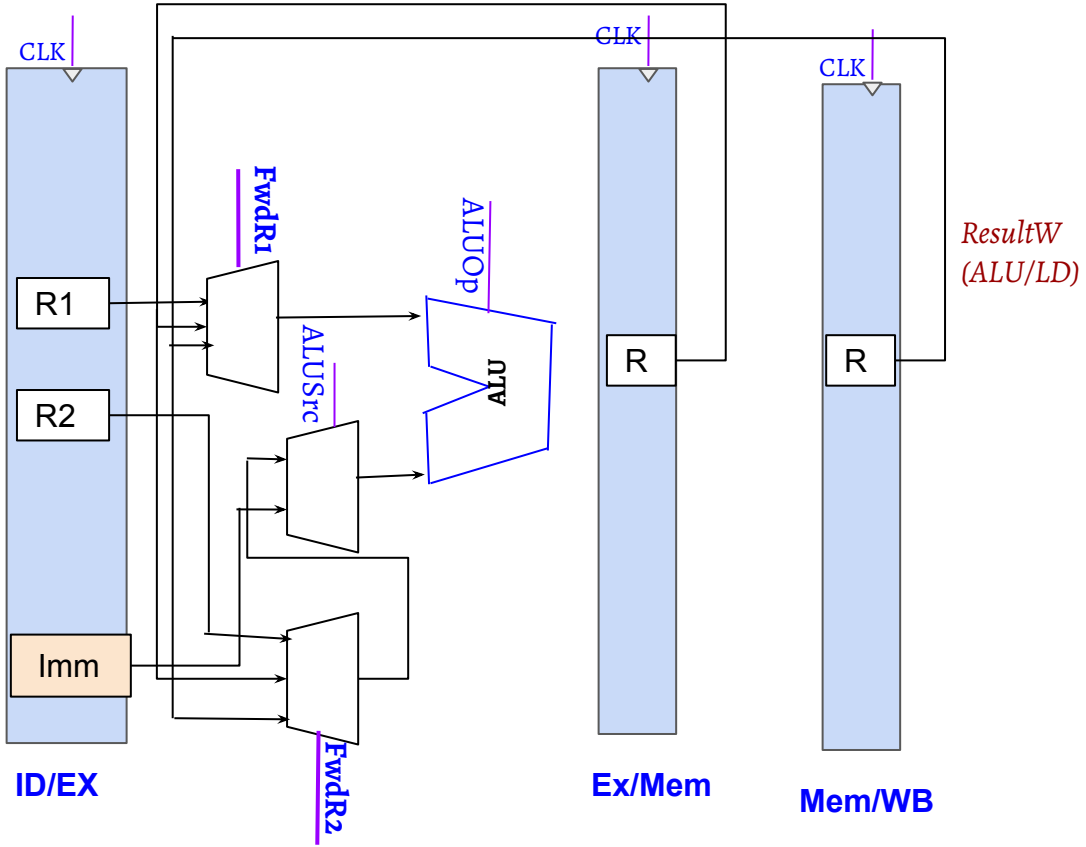

- Register **r1** value is already generated in CC3 (as ALU result)
- Can forward the value of R1 from the memory (CC4) and writeback stages (CC5) as inputs to the ALU
- How to implement?
- How to detect dependency?

# Bypass Network/Forwarding



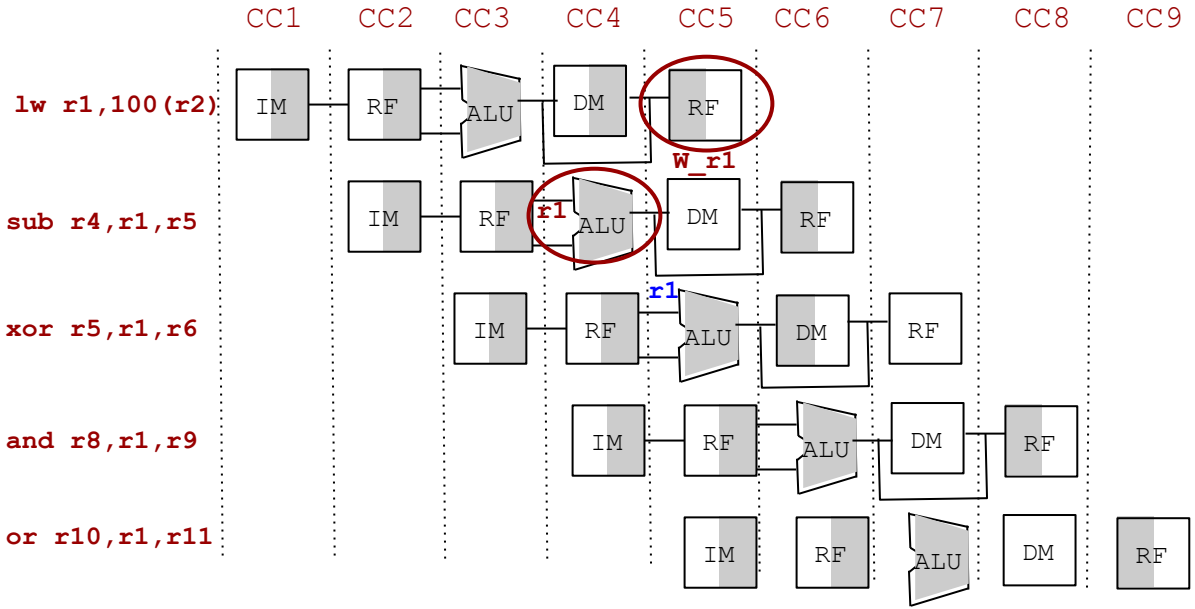

- How to implement?
- Forward the result of ALU output or write back result to the Ex stage, generate additional control signals to select them
- How to detect dependency?

# Bypass Network/Forwarding



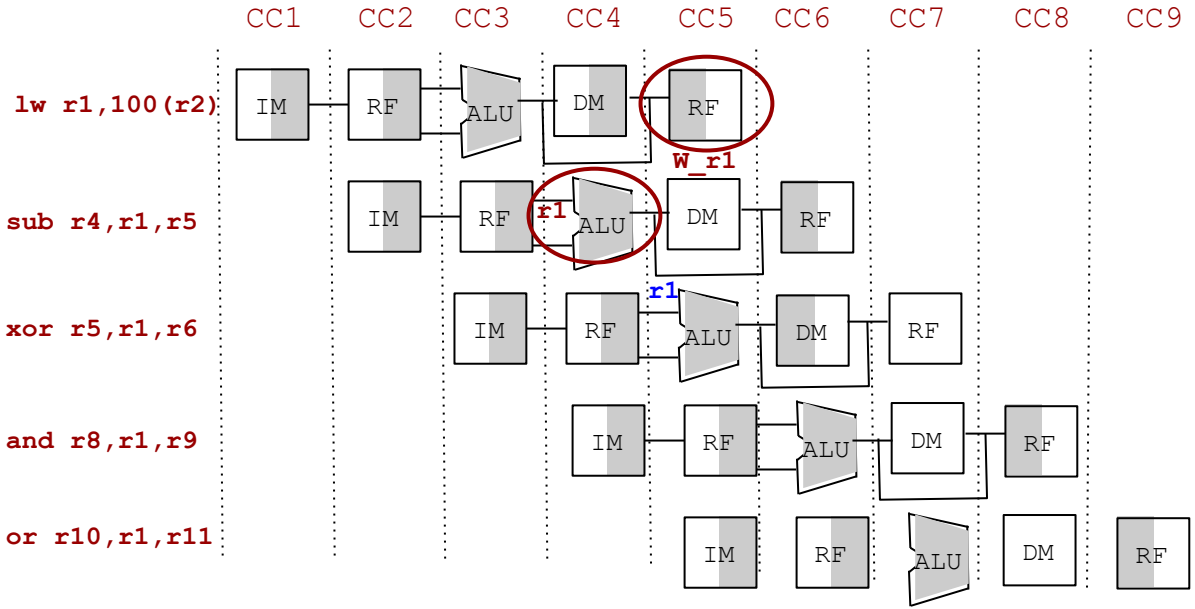

- How to implement?
- Forward the result of ALU output or write back result to the Ex stage, generate additional control signals to select them
- How to detect dependency?
- Additional control logic (hazard detection unit) generates control signals by comparing the source operand registers (in Ex stage) with the destination register (in Mem or WB stage)

# Does forwarding work always?




- Even with forwarding, **`sub`** stalls for one cycle
- What else can we do?

# Does forwarding work always?




- Even with forwarding, **`sub`** stalls for one cycle
- What else can we do?
- What if we execute an independent instruction after **`lw`**?

# Reordering to address data dependencies

- Consider a processor with a FPU which has the following stall behavior
  - Three cycles of stall for two dependent consecutive FP operations
  - One cycle stall for load of floating point followed by a FP operation
  - No stall required for two independent consecutive FP operations

**Program Order**

```
ldD f0,0(r1)
addD f2, f0, f1
subD f3, f2, f0
ld  r5,10(r1)
and r8,r5,r9
addD f5,f4,f7
add r3,r3,r2
```

One cycle

Three cycles

One cycle

How many stalls?

Pipeline stalls for five cycles. Can we avoid that?

# Reordering to address data dependencies

- Consider a processor with a FPU which has the following stall behavior
  - Three cycles of stall for two dependent consecutive FP operations
  - One cycle stall for load of floating point followed by a FP operation
  - No stall required for two independent consecutive FP operations

**Program Order**

```
ldD f0,0(r1)
addD f2, f0, f1
subD f3, f2, f0
ld  r5,10(r1)
and r8,r5,r9
addD f5,f4,f7
add r3,r3,r2
```

**Scheduling Order**

```
ldD f0,0(r1)
ld  r5,10(r1)
addD f2, f0, f1
addD f5,f4,f7
and r8,r5,r9
add r3,r3,r2
subD f3, f2, f0
```

How many stalls?

Pipeline stalls for five cycles. Can we avoid that? Yes, by carefully scheduling the instructions based on their data dependencies

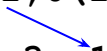
# Non-deterministic operations

- Consider a processor with three levels of cache with following round trip latencies
  - L1 hit: 1 cycle, L2 hit: 10 cycles, L3 hit: 30 cycles, Memory: 100 cycles

**Program Order**

```
a = a + 2000;

addi r4,r0,#2000
lw r1,0(r2)
add r3,r1,r4
sw r3, 0(r2)
```



Assuming `r2` contains the address of variable `a`, how many cycles of stall are expected?

# Non-deterministic operations

- Consider a processor with three levels of cache with following round trip latencies
  - L1 hit: 1 cycle, L2 hit: 10 cycles, L3 hit: 30 cycles, Memory: 100 cycles

**Program Order**

```
a = a + 2000;

addi r4,r0,#2000
lw r1,0(r2)
add r3,r1,r4
sw r3, 0(r2)
```

Assuming `r2` contains the address of variable `a`, how many cycles of stall are expected? Depends on where **a** resides at the time of instructions execution. Most of the times non-deterministic.

It would be better if reordering technique is flexible to handle these scenarios efficiently

# Why not static scheduling?

- Can not efficiently predict and handle unpredictable delays
  - A memory load may be served in different cycles at different times of execution
- Tight coupling with the underlying micro-architecture
  - Code generated for a particular machine may not be suitable for another machine
- Not all dependencies can be known at the compiler time
  - Memory dependencies, dependencies after dynamic linking

# Why not static scheduling?

- Can not efficiently predict and handle unpredictable delays
  - A memory load may be served in different cycles at different times of execution
- Tight coupling with the underlying micro-architecture
  - Code generated for a particular machine may not be suitable for another machine
- Not all dependencies can be known at the compiler time
  - Memory dependencies, dependencies after dynamic linking

Dynamic scheduling may avoid stalls and handle unpredictable delays

Keeping branch aside, are there any constraints/challenges?

# Why not static scheduling?

- Can not efficiently predict and handle unpredictable delays
  - A memory load may be served in different cycles at different times of execution
- Tight coupling with the underlying micro-architecture
  - Code generated for a particular machine may not be suitable for another machine
- Not all dependencies can be known at the compiler time
  - Memory dependencies, dependencies after dynamic linking

Dynamic scheduling may avoid stalls and handle unpredictable delays

Keeping branch aside, are there any constraints/challenges?

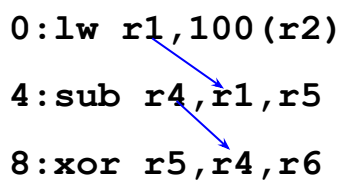
Let us re-examine the data dependence in presence of dynamic scheduling!

# Data dependance

- Data dependance: Instruction ***j*** is dependent on Instruction ***i*** if
  - Instruction ***i*** produces a results which is used by instruction ***j***
  - Instruction ***j*** is data dependent on ***k***, and instruction ***k*** is data dependent on instruction ***i***

# Data dependance

- Data dependance: Instruction ***j*** is dependent on Instruction ***i*** if
  - Instruction ***i*** produces a results which is used by instruction ***j***
  - Instruction ***j*** is data dependent on ***k***, and instruction ***k*** is data dependent on instruction ***i***

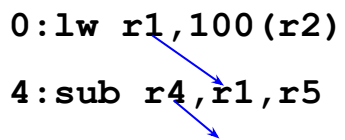



Instruction `sub` and `xor` are dependent on `lw`

How easy it is to determine data dependence?

# Data dependance

- Data dependance: Instruction ***j*** is dependent on Instruction ***i*** if
  - Instruction ***i*** produces a results which is used by instruction ***j***
  - Instruction ***j*** is data dependent on ***k***, and instruction ***k*** is data dependent on instruction ***i***




Instruction `sub` and `xor` are dependent on `lw`

- Detecting register data dependence is comparatively easier than detecting memory dependence
- Example: `r1=1000 r2=900`
  `sw r5, 110(r2)` and `lw r7, 10(r1)` are data dependent

# Name dependance

- Name dependance occurs when two instructions use same memory location or register but there is no data flow between the two instructions. Assuming instruction ***i*** precedes instruction ***j*** in program order,
  - Instruction ***i*** is name dependent on ***j*,** if ***j*** writes to a register/memory location which is read by instruction ***i*** (Anti-dependence)
  - Instruction ***j*** and ***i*** write to the same register/memory location (Output dependence)

```
0:lw r1,100(r2)
4:sub r4,r1,r5
8:xor r1,r1,r6
```

- Why worry about name dependence if they are not true data dependencies?
- How can we address name dependence?

# Name dependance

- Name dependance occurs when two instructions use same memory location or register but there is no data flow between the two instructions. Assuming instruction ***i*** precedes instruction ***j*** in program order,
  - Instruction ***i*** is name dependent on ***j*,** if ***j*** writes to a register/memory location which is read by instruction ***i*** (Anti-dependence)
  - Instruction ***j*** and ***i*** write to the same register/memory location (Output dependence)

```
0:lw r1,100(r2)
4:sub r4,r1,r5
8:xor r1,r1,r6
```

- Why worry about name dependence if they are not true data dependencies? It matters when instructions are not executed in program order
- How can we address name dependence? For registers, register renaming can be used

# Program order and hazards

- Objective of any software/hardware technique is to preserve the program order
- Dependent instructions should be executed such that the output matches the sequential execution order (exploiting maximum parallelism possible)

```
0:lw r1,100(r2)
4:sub r4,r1,r5
```

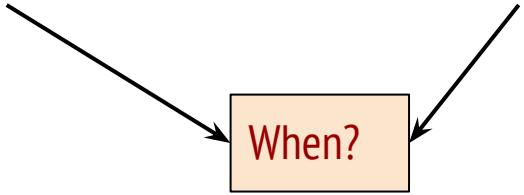
**RAW**
**(True data dependence)**

```
0:lw r1,100(r2)
4:sub r1,r2,r5
```

**WAW**
**(Output dependence)**

```
0:lw r1,100(r2)
4:sub r2,r4,r5
```

**WAR**
**(Name dependence)**



# Program order and hazards

- Objective of any software/hardware technique is to preserve the program order
- Dependent instructions should be executed such that the output matches the sequential execution order (exploiting maximum parallelism possible)

```
0:lw r1,100(r2)
4:sub r4,r1,r5
```

**RAW**
**(True data dependence)**

```
0:lw r1,100(r2)
4:sub r1,r2,r5
```

**WAW**
**(Output dependence)**

```
0:lw r1,100(r2)
4:sub r2,r4,r5
```

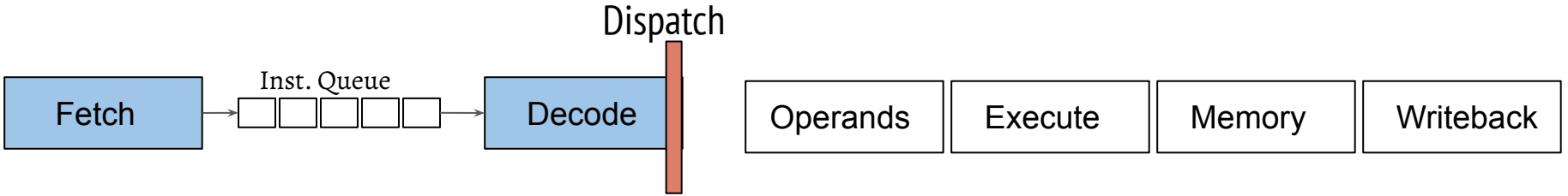
**WAR**
**(Name dependence)**

- Instruction reordering
- Write/read is pipelined or their order is not preserved

# Dynamic Scheduling: Overall scheme

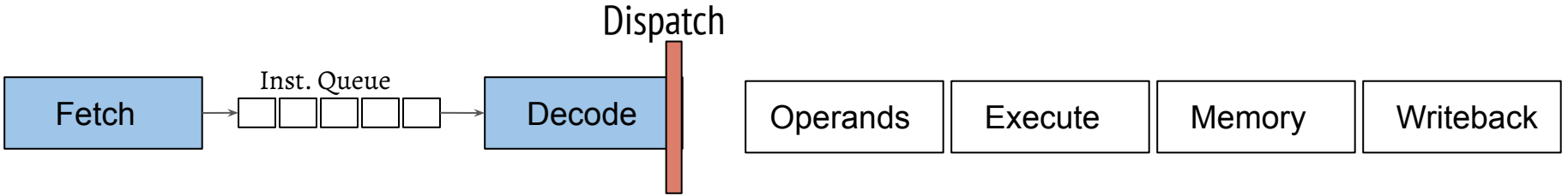

- Considering the five stage pipeline, in which stage, the dynamic scheduling should take place and why?
- What are the changes to the pipeline structure?
- What are the relevant design questions?

# Dynamic Scheduling: Overall scheme



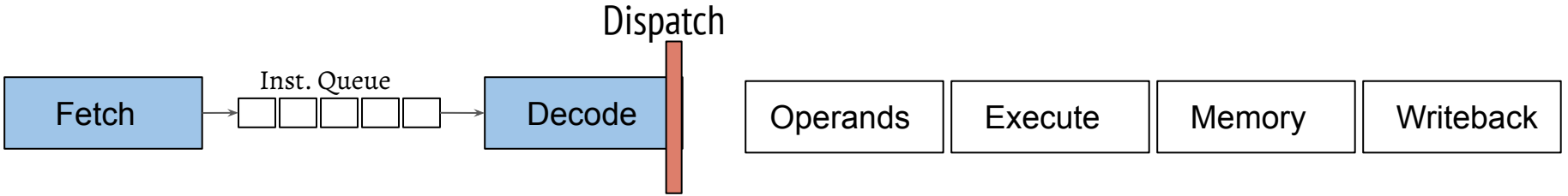

- Considering the five stage pipeline, in which stage, the dynamic scheduling should take place and why? Decode; dependency can be determined only after decoding the instructions
- What are the changes to the pipeline structure? Decode and RF access should be two different stages. Typically, an instruction queue between the fetch and decode.
- What are the relevant design questions?

# Dynamic Scheduling: Overall scheme



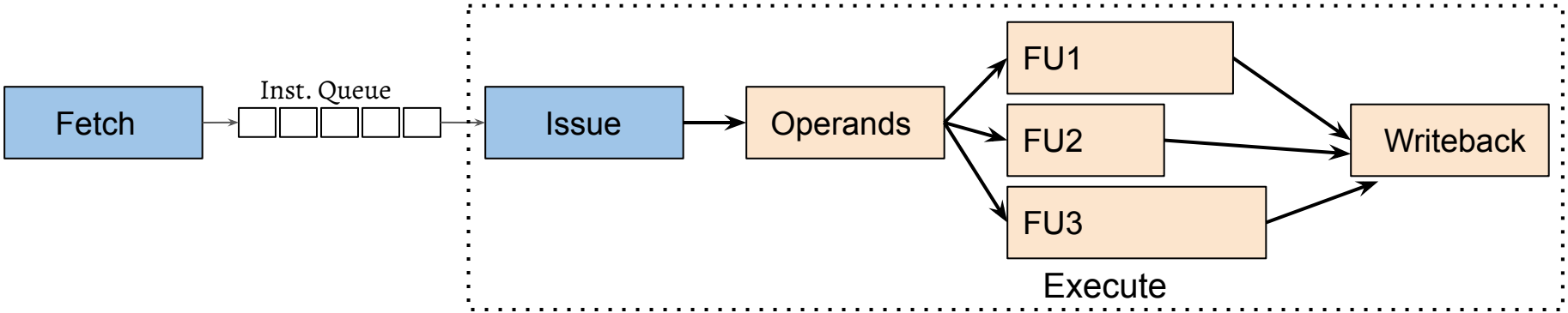

- Considering the five stage pipeline, in which stage, the dynamic scheduling should take place and why? Decode; dependency can be determined only after decoding the instructions
- What are the changes to the pipeline structure? Decode and RF access should be two different stages. Typically, an instruction queue between the fetch and decode.
- What are the relevant design questions?
  - What is the size of instruction queue? Importantly, can instructions across branches be part of the scheduling decision?
  - What to do with instructions with different dependencies? {Stall} or {schedule but handle during later stages}
  - How many instructions to schedule?

# Dynamic Scheduling: Overall scheme



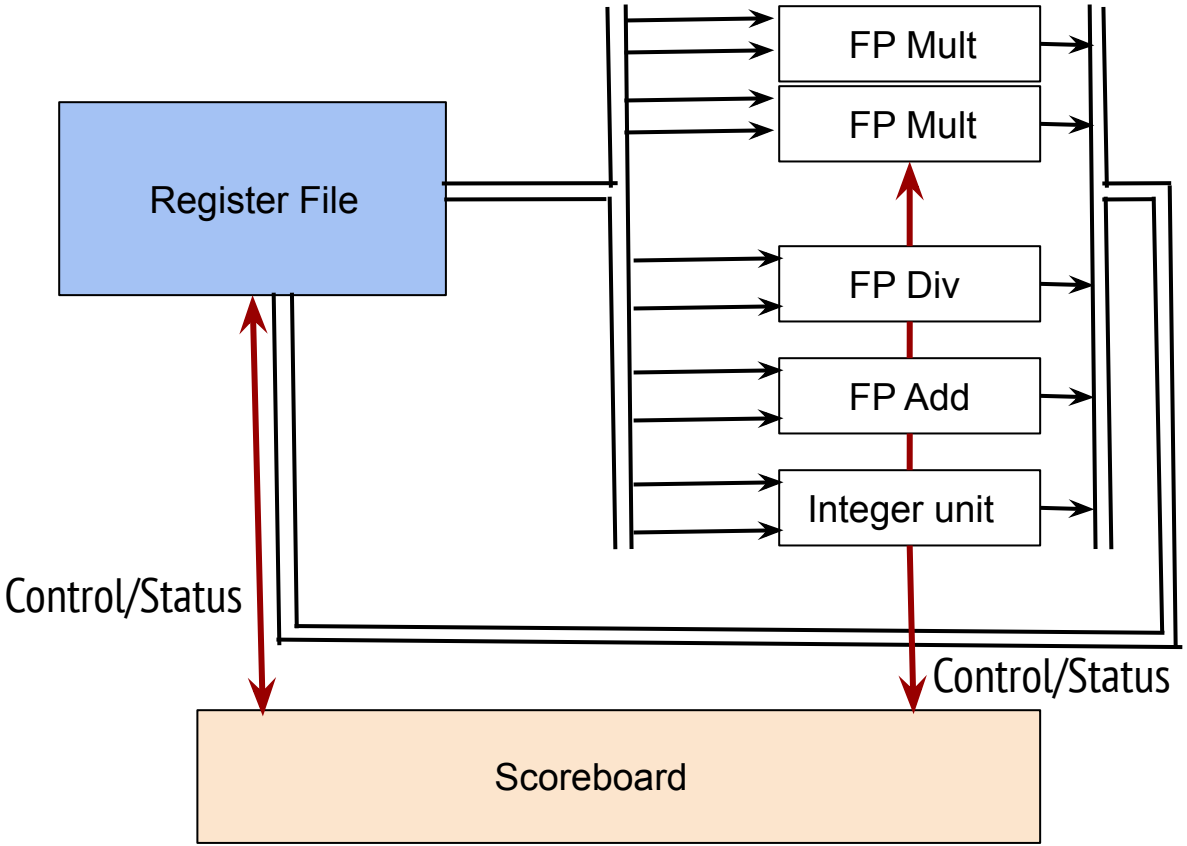

- What is the size of instruction queue? Importantly, can instructions across branches be part of the scheduling decision? For improved performance, a design should allow instructions across branches. Issues to tackle: branch misprediction {Will revisit}
- What to do with instructions with different dependencies? Stall or schedule but handle during later stages? Advancing the instructions as much seems to be a good idea, but locking resources while waiting for operands (RAW) can be detrimental. Advancing instructions with WAR/WAW hazards should make sure of program correctness.
- How many instructions to schedule? More FUs/access ports to RF/Mem, instruction/FU state maintenance overheads are not the only issues here! Penalty of doing useless work and nullifying their impacts because of control hazards is also an important concern

# Dynamic Scheduling: Scoreboard




- Assumptions
  - Multiple/pipelined functional units with different execution lengths
  - Schedule (or issue) instructions from the same basic block
  - Multiport register file (why?)
  - No forwarding/bypass network
  - Memory LD/ST is one cycle and requires an adder (for simplifying our discussion, we will remove this soon)

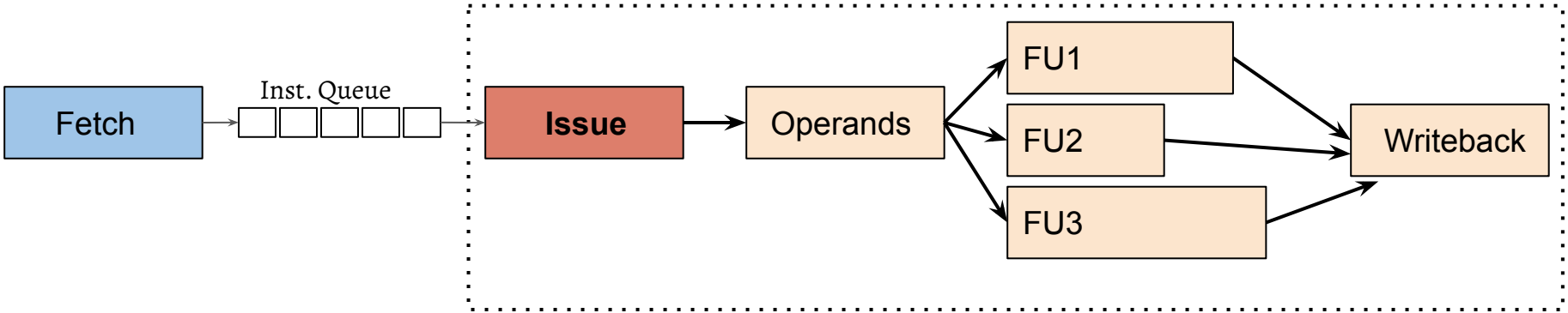
# Basic structure of an example scoreboard




Integer unit performs all ALU operations and operations for memory address calculation

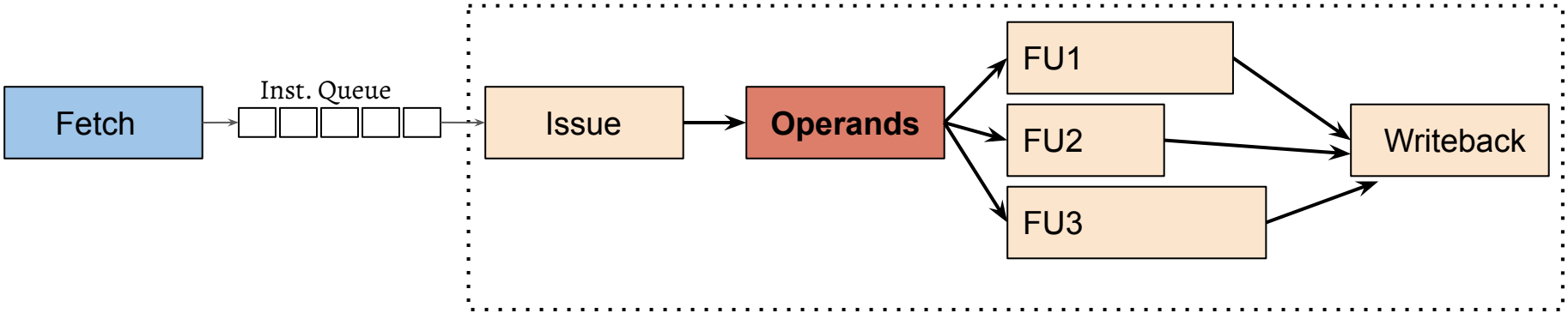
#of concurrent access to register file handled as a structural hazard issue

# Scoreboard: Issue



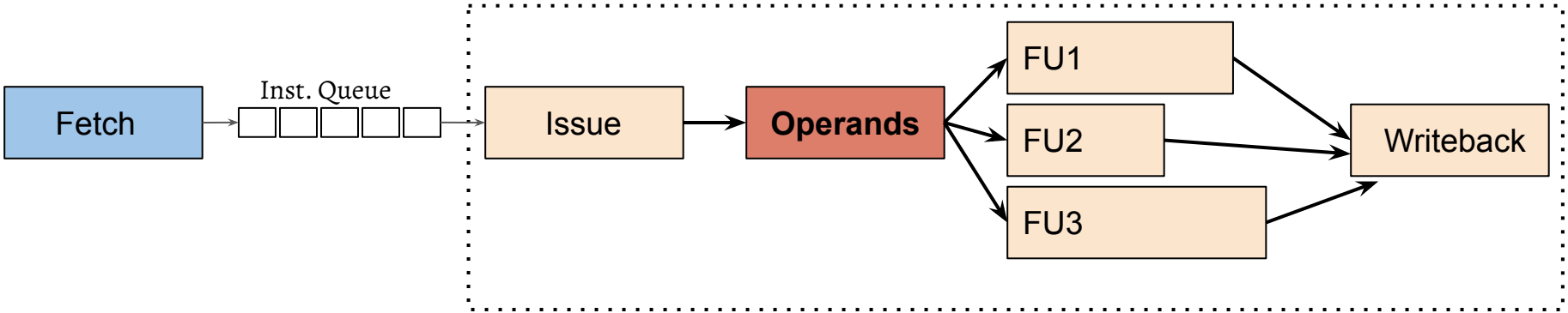

- Decode instruction and check for structural hazards
  - Need information regarding current usage of functional units (part of scoreboard)
  - Stop issue till structural hazard is resolved
- Stall the instruction if its output dependant on any instruction currently in execution
  - WAW is taken care of in the issue stage
- All issued instructions are tracked using the scoreboard

# Scoreboard: Read operands (dispatch)



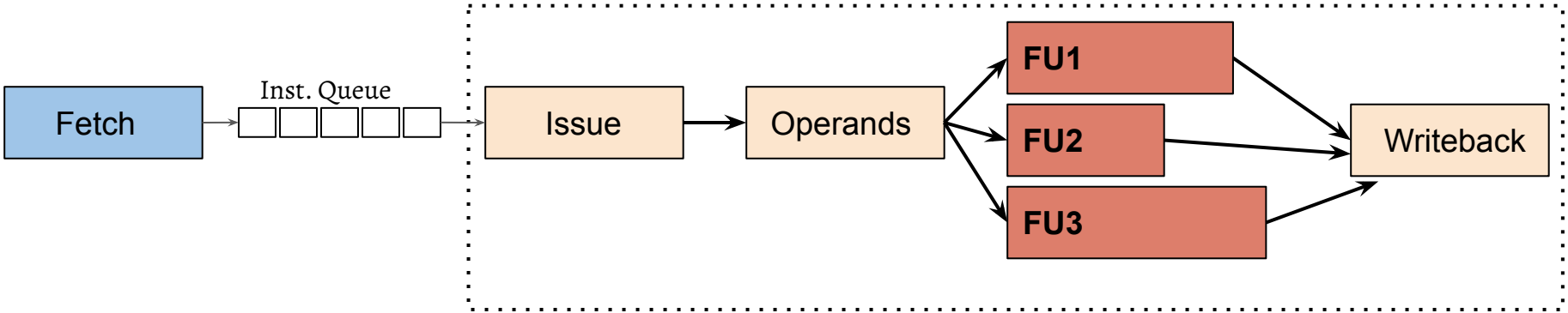

- Instruction with true data dependence stall
  - Information regarding waiting instructions maintained in the scoreboard
  - When source operands are available, the dependent instructions start execution (a.k.a. dispatched OoO, WAR is a possibility!)
- What should be the #of read ports in RF?

# Scoreboard: Read operands (dispatch)



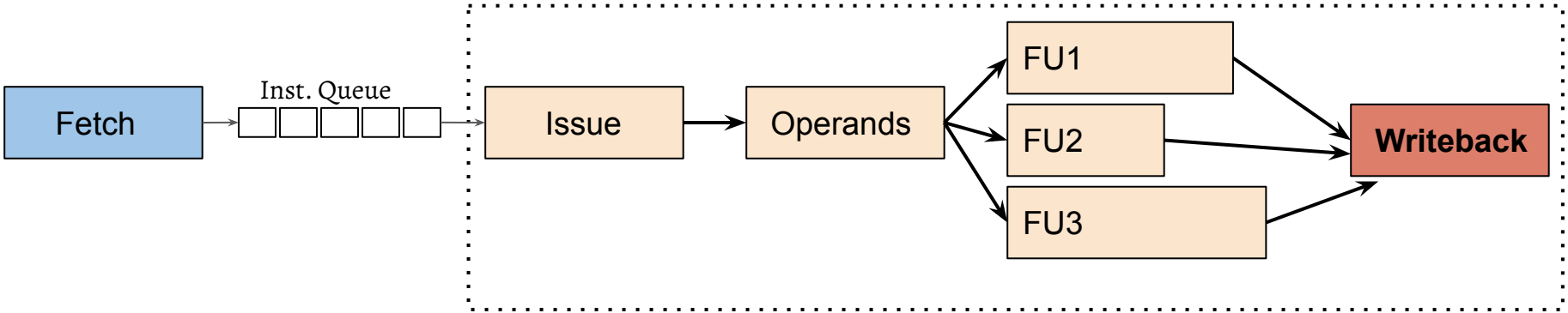

- Instruction with true data dependence stall
  - Information regarding waiting instructions maintained in the scoreboard
  - When source operands are available, the dependent instructions start execution (a.k.a. dispatched OoO)
- What should be the #of read ports in RF? 2 * Number of functional units (assuming functional units are marked free after execute)

# Scoreboard: Execute




- FU starts executing the instruction (can consume variable #of cycles)
- Notifies the scoreboard on completion so that the scoreboard can update the state of the FU and the instruction

# Scoreboard: Write result




- Scoreboard checks for WAR hazard for the completed instruction and stalls if required
- All instructions clear of WAR can perform write back (more than one possible)
- Subtle issue: When should the FU is considered free, end of execution or after writing results?

# Structure of a Scoreboard

Instruction status

| Issue | ReadOp | Execute | WriteRes |
|---|---|---|---|
| | | | |
| | | | |

FU status

| FUID | Busy | Op | Fi | Fj Fk | Qj Qk | Rj Rk |
|---|---|---|---|---|---|---|
| | | | | | | |
| | | | | | | |

Register result status

| | F0 | F1 | F2 | … | Fn |
|---|---|---|---|---|---|
| FU | | | | | |

Busy: True/False Op: Operation Fi: Destination register

FjFk: Source register numbers

QjQk: Functional units producing FjFk

RjRk: FjFk are ready but not yet read

- Instruction status: Status of every instruction admitted to the scoreboard
- Functional unit status: Indicates the state of the functional unit
- Register result status: Functional units used by active instructions which will write to a given register

# A snapshot representation of the scoreboard

Instruction status

| Inst | Issue | ReadOp | Execute | WriteRes |
|---|---|---|---|---|
| I1 | 1 | 2 | 3 | |
| I2 | 2 | 3 | | |

FU status

| FUID | Busy | Op | Fi | Fj Fk | Qj Qk | Rj Rk |
|---|---|---|---|---|---|---|
| INT | 1 | Add | f7 | r2 _ | _ _ | No No |
| FA | 1 | Sub | f3 | f2 f4 | _ _ | Yes Yes |

Register result status

| | F0 | F1 | F2 | F3 | F7 |
|---|---|---|---|---|---|
| FU | | | | FA | INT |

| | | I3 | I2 | I1 |
|---|---|---|---|---|

Instruction Queue

Clock Cycle

- Snapshot at the beginning of 4th clock cycle
- Clock cycle when the instruction entered that stage is shown (for our understanding)
- I3 is considered next to be issued

# Example basic block

```
I1: lD f6,34(r2)
I2: lD f2,45(r3)
I3: mulD f0,f2,f4
I4: subD f8,f6,f2
I5: divD f10,f0,f6
I6: addD f6,f8,f2
```

- Execution time in cycles: Integer Op = 1, FP add = 2, FP multiply = 10 and FP divide = 40
- Possible hazards
  - Structural: `{I1,I2}{I4,I6}`
  - RAW: `{I1,I4},{I1,I5},{I2,I3},{I2,I4},{I2,I6}{I3,I5}{I4,I6}`
  - WAR: `{I4,I6},{I5,I6}`

# Example: Cycle 0

Instruction status

| Inst | Issue | ReadOp | Execute | WriteRes |
|---|---|---|---|---|
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

FU status

| FUID | Busy | Op | Fi | Fj Fk | Qj Qk | Rj Rk |
|---|---|---|---|---|---|---|
| INT | 0 | | | | | |
| FM1 | 0 | | | | | |
| FM2 | 0 | | | | | |
| FA | 0 | | | | | |
| FD | 0 | | | | | |

| | | | | |
|---|---|---|---|---|
| | | | | I1 |

Instruction Queue

```
I1: lD f6,34(r2)
I2: lD f2,45(r3)
I3: mulD f0,f2,f4
I4: subD f8,f6,f2
I5: divD f10,f0,f6
I6: addD f6,f8,f2
```

Register result status

| | F0 | F1 | F2 | F3 | F4 | F5 | F6 | F7 | F8 |
|---|---|---|---|---|---|---|---|---|---|
| FU | | | | | | | | | |

# Example: Cycle 1

Instruction status

| Inst | Issue | ReadOp | Execute | WriteRes |
|---|---|---|---|---|
| I1 | 1 | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

FU status

| FUID | Busy | Op | Fi | Fj Fk | Qj Qk | Rj Rk |
|---|---|---|---|---|---|---|
| INT | 1 | add_l | f6 | r2 _ | _ _ | Yes _ |
| FM1 | 0 | | | | | |
| FM2 | 0 | | | | | |
| FA | 0 | | | | | |
| FD | 0 | | | | | |

| | | | | |
|---|---|---|---|---|
| | | | I2 | I1 |

Instruction Queue

```
I1: lD f6,34(r2)
I2: lD f2,45(r3)
I3: mulD f0,f2,f4
I4: subD f8,f6,f2
I5: divD f10,f0,f6
I6: addD f6,f8,f2
```

Register result status

| | f0 | f1 | f2 | f3 | f4 | f5 | f6 | f7 | f8 |
|---|---|---|---|---|---|---|---|---|---|
| FU | | | | | | | INT | | |

# Example: Cycle 2

Instruction status

| Inst | Issue | ReadOp | Execute | WriteRes |
|---|---|---|---|---|
| I1 | 1 | 2 | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

FU status

| FUID | Busy | Op | Fi | Fj Fk | Qj Qk | Rj Rk |
|---|---|---|---|---|---|---|
| INT | 1 | add_l | f6 | r2 _ | _ _ | Yes _ |
| FM1 | 0 | | | | | |
| FM2 | 0 | | | | | |
| FA | 0 | | | | | |
| FD | 0 | | | | | |

| | | | | |
|---|---|---|---|---|
| | | I3 | I2 | I1 |

Instruction Queue

```
I1: lD f6,34(r2)
I2: lD f2,45(r3)
I3: mulD f0,f2,f4
I4: subD f8,f6,f2
I5: divD f10,f0,f6
I6: addD f6,f8,f2
```

Register result status

| | f0 | f1 | f2 | f3 | f4 | f5 | f6 | f7 | f8 |
|---|---|---|---|---|---|---|---|---|---|
| FU | | | | | | | INT | | |

Structural hazard, can't issue I2

# Example: Cycle 3

Instruction status

| Inst | Issue | ReadOp | Execute | WriteRes |
|---|---|---|---|---|
| I1 | 1 | 2 | 3 | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

FU status

| FUID | Busy | Op | Fi | Fj Fk | Qj Qk | Rj Rk |
|---|---|---|---|---|---|---|
| INT | 1 | add_l | f6 | r2 _ | _ _ | No _ |
| FM1 | 0 | | | | | |
| FM2 | 0 | | | | | |
| FA | 0 | | | | | |
| FD | 0 | | | | | |

| | I4 | I3 | I2 | I1 |
|---|---|---|---|---|

Instruction Queue

```
I1: lD f6,34(r2)
I2: lD f2,45(r3)
I3: mulD f0,f2,f4
I4: subD f8,f6,f2
I5: divD f10,f0,f6
I6: addD f6,f8,f2
```

Register result status

| | f0 | f1 | f2 | f3 | f4 | f5 | f6 | f7 | f8 |
|---|---|---|---|---|---|---|---|---|---|
| FU | | | | | | | INT | | |

Structural hazard, can't issue I2

# Example: Cycle 4

Instruction status

| Inst | Issue | ReadOp | Execute | WriteRes |
|---|---|---|---|---|
| *I1* | *1* | *2* | *3* | *4* |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

FU status

| FUID | Busy | Op | Fi | Fj Fk | Qj Qk | Rj Rk |
|---|---|---|---|---|---|---|
| INT | 0 | | | | | |
| FM1 | 0 | | | | | |
| FM2 | 0 | | | | | |
| FA | 0 | | | | | |
| FD | 0 | | | | | |

| I5 | I4 | I3 | I2 | I1 |
|---|---|---|---|---|

Instruction Queue

```
I1: lD f6,34(r2)
I2: lD f2,45(r3)
I3: mulD f0,f2,f4
I4: subD f8,f6,f2
I5: divD f10,f0,f6
I6: addD f6,f8,f2
```

Register result status

| | f0 | f1 | f2 | f3 | f4 | f5 | f6 | f7 | f8 |
|---|---|---|---|---|---|---|---|---|---|
| FU | | | | | | | INT | | |

INT can be used in the next CC, f6 available in next CC

# Example: Cycle 5

Instruction status

| Inst | Issue | ReadOp | Execute | WriteRes |
|---|---|---|---|---|
| *I1* | *1* | *2* | *3* | *4* |
| I2 | 5 | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

FU status

| FUID | Busy | Op | Fi | Fj Fk | Qj Qk | Rj Rk |
|---|---|---|---|---|---|---|
| INT | 1 | add_l | f2 | r3 _ | _ _ | Yes _ |
| FM1 | 0 | | | | | |
| FM2 | 0 | | | | | |
| FA | 0 | | | | | |
| FD | 0 | | | | | |

| I6 | I5 | I4 | I3 | I2 |
|---|---|---|---|---|

Instruction Queue

```
I1: lD f6,34(r2)
I2: lD f2,45(r3)
I3: mulD f0,f2,f4
I4: subD f8,f6,f2
I5: divD f10,f0,f6
I6: addD f6,f8,f2
```

Register result status

| | f0 | f1 | f2 | f3 | f4 | f5 | f6 | f7 | f8 |
|---|---|---|---|---|---|---|---|---|---|
| FU | | | INT | | | | | | |

Single Issue, load issued

# Example: Cycle 6

## Instruction status

| Inst | Issue | ReadOp | Execute | WriteRes |
|---|---|---|---|---|
| *I1* | *1* | *2* | *3* | *4* |
| I2 | 5 | 6 | | |
| I3 | 6 | | | |
| | | | | |
| | | | | |
| | | | | |

## FU status

| FUID | Busy | Op | Fi | Fj Fk | Qj Qk | Rj Rk |
|---|---|---|---|---|---|---|
| INT | 1 | add_l | f2 | r3 _ | _ _ | Yes _ |
| FM1 | 1 | Mul | f0 | f2 f4 | INT _ | No Yes |
| FM2 | 0 | | | | | |
| FA | 0 | | | | | |
| FD | 0 | | | | | |

| I6 | I5 | I4 | I3 | I2 |
|---|---|---|---|---|

Instruction Queue

*I1: lD f6,34(r2)*

I2: lD f2,45(r3)

I3: mulD f0,f2,f4

I4: subD f8,f6,f2

I5: divD f10,f0,f6

I6: addD f6,f8,f2

## Register result status

| | f0 | f1 | f2 | f3 | f4 | f5 | f6 | f7 | f8 |
|---|---|---|---|---|---|---|---|---|---|
| FU | FM1 | | INT | | | | | | |

# Example: Cycle 7

## Instruction status

| Inst | Issue | ReadOp | Execute | WriteRes |
|---|---|---|---|---|
| *I1* | *1* | *2* | *3* | *4* |
| I2 | 5 | 6 | 7 | |
| I3 | 6 | | | |
| I4 | 7 | | | |
| | | | | |
| | | | | |

## FU status

| FUID | Busy | Op | Fi | Fj Fk | Qj Qk | Rj Rk |
|---|---|---|---|---|---|---|
| INT | 1 | add_l | f2 | r3 _ | _ _ | No _ |
| FM1 | 1 | Mul | f0 | f2 f4 | INT _ | No Yes |
| FM2 | 0 | | | | | |
| FA | 1 | Sub | f8 | f6 f2 | _ INT | Yes No |
| FD | 0 | | | | | |

| I6 | I5 | I4 | I3 | I2 |
|---|---|---|---|---|

Instruction Queue

```
I1: lD f6,34(r2)
I2: lD f2,45(r3)
I3: mulD f0,f2,f4
I4: subD f8,f6,f2
I5: divD f10,f0,f6
I6: addD f6,f8,f2
```

## Register result status

| | f0 | f1 | f2 | f3 | f4 | f5 | f6 | f7 | f8 |
|---|---|---|---|---|---|---|---|---|---|
| FU | FM1 | | INT | | | | | | FA |

# Example: Cycle 8 (at end)

Instruction status

| Inst | Issue | ReadOp | Execute | WriteRes |
|---|---|---|---|---|
| *I1* | *1* | *2* | *3* | *4* |
| I2 | 5 | 6 | 7 | 8 |
| I3 | 6 | | | |
| I4 | 7 | | | |
| I5 | 8 | | | |
| | | | | |

FU status

| FUID | Busy | Op | Fi | Fj Fk | Qj Qk | Rj Rk |
|---|---|---|---|---|---|---|
| INT | 0 | | | | | |
| FM1 | 1 | Mul | f0 | f2 f4 | INT _ | Yes Yes |
| FM2 | 0 | | | | | |
| FA | 1 | Sub | f8 | f6 f2 | _ INT | Yes Yes |
| FD | 1 | Div | f10 | f0 f6 | FM1 _ | No Yes |

| I6 | I5 | I4 | I3 | I2 |
|---|---|---|---|---|

Instruction Queue

*I1: lD f6,34(r2)*

I2: lD f2,45(r3)

I3: mulD f0,f2,f4

I4: subD f8,f6,f2

I5: divD f10,f0,f6

I6: addD f6,f8,f2

Register result status

| | f0 | f10 | f2 | f3 | f4 | f5 | f6 | f7 | f8 |
|---|---|---|---|---|---|---|---|---|---|
| FU | FM1 | FD | | | | | | | FA |

f2 available in next CC. I3 and I4 can read operands

# Example: Cycle 9

## Instruction status

| Inst | Issue | ReadOp | Execute | WriteRes |
|---|---|---|---|---|
| *I1* | *1* | *2* | *3* | *4* |
| I2 | 5 | 6 | 7 | 8 |
| I3 | 6 | 9 | | |
| I4 | 7 | 9 | | |
| I5 | 8 | | | |
| | | | | |

## FU status

| FUID | Busy | Op | Fi | Fj Fk | Qj Qk | Rj Rk |
|---|---|---|---|---|---|---|
| INT | 0 | | | | | |
| FM1 | 1 | Mul | f0 | f2 f4 | _ _ | Yes Yes |
| FM2 | 0 | | | | | |
| FA | 1 | Sub | f8 | f6 f2 | _ _ | Yes Yes |
| FD | 1 | Div | f10 | f0 f6 | FM1 _ | No Yes |

| | | | | |
|---|---|---|---|---|
| | I6 | I5 | I4 | I3 |

Instruction Queue

*I1: lD f6,34(r2)*

*I2: lD f2,45(r3)*

I3: mulD f0,f2,f4

I4: subD f8,f6,f2

I5: divD f10,f0,f6

I6: addD f6,f8,f2

## Register result status

| | f0 | f10 | f2 | f3 | f4 | f5 | f6 | f7 | f8 |
|---|---|---|---|---|---|---|---|---|---|
| FU | FM1 | FD | | | | | | | FA |

# Example: Cycle 10

## Instruction status

| Inst | Issue | ReadOp | Execute | WriteRes |
|---|---|---|---|---|
| *I1* | *1* | *2* | *3* | *4* |
| I2 | 5 | 6 | 7 | 8 |
| I3 | 6 | 9 | 10(r9) | |
| I4 | 7 | 9 | 10(r1) | |
| I5 | 8 | | | |
| | | | | |

## FU status

| FUID | Busy | Op | Fi | Fj Fk | Qj Qk | Rj Rk |
|---|---|---|---|---|---|---|
| INT | 0 | | | | | |
| FM1 | 1 | Mul | f0 | f2 f4 | _ _ | No No |
| FM2 | 0 | | | | | |
| FA | 1 | Sub | f8 | f6 f2 | _ _ | No No |
| FD | 1 | Div | f10 | f0 f6 | FM1 _ | No Yes |

| | I6 | I5 | I4 | I3 |
|---|---|---|---|---|

Instruction Queue

*I1: lD f6,34(r2)*

*I2: lD f2,45(r3)*

I3: mulD f0,f2,f4

I4: subD f8,f6,f2

I5: divD f10,f0,f6

I6: addD f6,f8,f2

## Register result status

| | f0 | f10 | f2 | f3 | f4 | f5 | f6 | f7 | f8 |
|---|---|---|---|---|---|---|---|---|---|
| FU | FM1 | FD | | | | | | | FA |

# Example: Cycle 11

Instruction status

| Inst | Issue | ReadOp | Execute | WriteRes |
|---|---|---|---|---|
| *I1* | *1* | *2* | *3* | ***4*** |
| I2 | 5 | 6 | **7** | **8** |
| I3 | **6** | **9** | **11(r8)** | |
| I4 | **7** | **9** | **11** | |
| I5 | 8 | | | |
| | | | | |

FU status

| FUID | Busy | Op | Fi | Fj Fk | Qj Qk | Rj Rk |
|---|---|---|---|---|---|---|
| INT | 0 | | | | | |
| FM1 | 1 | Mul | f0 | f2 f4 | _ _ | No No |
| FM2 | 0 | | | | | |
| FA | 1 | Sub | f8 | f6 f2 | _ _ | No No |
| FD | 1 | Div | f10 | f0 f6 | FM1 _ | No Yes |

| | I6 | I5 | I4 | I3 |
|---|---|---|---|---|

Instruction Queue

*I1: lD f6,34(r2)*

*I2: lD f2,45(r3)*

I3: mulD f0,f2,f4

I4: subD f8,f6,f2

I5: divD f10,f0,f6

I6: addD f6,f8,f2

Register result status

| | f0 | f10 | f2 | f3 | f4 | f5 | f6 | f7 | f8 |
|---|---|---|---|---|---|---|---|---|---|
| FU | FM1 | FD | | | | | | | FA |

I4 finished execution, I6 can not be issued yet

# Example: Cycle 12 (at end)

Instruction status

| Inst | Issue | ReadOp | Execute | WriteRes |
|---|---|---|---|---|
| *I1* | *1* | *2* | *3* | ***4*** |
| I2 | 5 | 6 | **7** | **8** |
| I3 | **6** | **9** | **12(r7)** | |
| I4 | **7** | **9** | **11** | **12** |
| I5 | 8 | | | |
| | | | | |

FU status

| FUID | Busy | Op | Fi | Fj Fk | Qj Qk | Rj Rk |
|---|---|---|---|---|---|---|
| INT | 0 | | | | | |
| FM1 | 1 | Mul | f0 | f2 f4 | _ _ | No No |
| FM2 | 0 | | | | | |
| FA | 0 | | | | | |
| FD | 1 | Div | f10 | f0 f6 | FM1 _ | No Yes |

| | I6 | I5 | I4 | I3 |
|---|---|---|---|---|

Instruction Queue

*I1: lD f6,34(r2)*

*I2: lD f2,45(r3)*

I3: mulD f0,f2,f4

I4: subD f8,f6,f2

I5: divD f10,f0,f6

I6: addD f6,f8,f2

Register result status

| | f0 | f10 | f2 | f3 | f4 | f5 | f6 | f7 | f8 |
|---|---|---|---|---|---|---|---|---|---|
| FU | FM1 | FD | | | | | | | |

I4 finished execution, I6 can be issued next CC

# Example: Cycle 13

## Instruction status

| Inst | Issue | ReadOp | Execute | WriteRes |
|---|---|---|---|---|
| *I1* | *1* | *2* | *3* | *4* |
| I2 | 5 | 6 | 7 | 8 |
| I3 | 6 | 9 | 13(r6) | |
| I4 | 7 | 9 | 11 | 12 |
| I5 | 8 | | | |
| I6 | 13 | | | |

## FU status

| FUID | Busy | Op | Fi | Fj Fk | Qj Qk | Rj Rk |
|---|---|---|---|---|---|---|
| INT | 0 | | | | | |
| FM1 | 1 | Mul | f0 | f2 f4 | _ _ | No No |
| FM2 | 0 | | | | | |
| FA | 1 | Add | f6 | f8 f2 | _ _ | Yes Yes |
| FD | 1 | Div | f10 | f0 f6 | FM1 _ | No Yes |

| | | | | |
|---|---|---|---|---|
| | I6 | I5 | | I3 |

Instruction Queue

*I1: lD f6,34(r2)*

*I2: lD f2,45(r3)*

I3: mulD f0,f2,f4

*I4: subD f8,f6,f2*

I5: divD f10,f0,f6

I6: addD f6,f8,f2

## Register result status

| | f0 | f10 | f2 | f3 | f4 | f5 | f6 | f7 | f8 |
|---|---|---|---|---|---|---|---|---|---|
| FU | FM1 | FD | | | | | FA | | |

# Example: Cycle 14

## Instruction status

| Inst | Issue | ReadOp | Execute | WriteRes |
|---|---|---|---|---|
| *I1* | *1* | *2* | *3* | ***4*** |
| I2 | 5 | 6 | **7** | **8** |
| I3 | **6** | **9** | **14(r5)** | |
| I4 | **7** | **9** | **11** | **12** |
| I5 | 8 | | | |
| I6 | **13** | **14** | | |

## FU status

| FUID | Busy | Op | Fi | Fj Fk | Qj Qk | Rj Rk |
|---|---|---|---|---|---|---|
| INT | 0 | | | | | |
| FM1 | 1 | Mul | f0 | f2 f4 | _ _ | No No |
| FM2 | 0 | | | | | |
| FA | 1 | Add | f6 | f8 f2 | _ _ | Yes Yes |
| FD | 1 | Div | f10 | f0 f6 | FM1 _ | No Yes |

| | | | | |
|---|---|---|---|---|
| | I6 | I5 | | I3 |

Instruction Queue

*I1: lD f6,34(r2)*

*I2: lD f2,45(r3)*

**I3: mulD f0,f2,f4**

*I4: subD f8,f6,f2*

**I5: divD f10,f0,f6**

**I6: addD f6,f8,f2**

## Register result status

| | f0 | f10 | f2 | f3 | f4 | f5 | f6 | f7 | f8 |
|---|---|---|---|---|---|---|---|---|---|
| FU | FM1 | FD | | | | | FA | | |

# Example: Cycle 15

## Instruction status

| Inst | Issue | ReadOp | Execute | WriteRes |
|---|---|---|---|---|
| *I1* | *1* | *2* | *3* | *4* |
| I2 | 5 | 6 | 7 | 8 |
| I3 | 6 | 9 | 15(r4) | |
| I4 | 7 | 9 | 11 | 12 |
| I5 | 8 | | | |
| I6 | 13 | 14 | 15(r1) | |

## FU status

| FUID | Busy | Op | Fi | Fj Fk | Qj Qk | Rj Rk |
|---|---|---|---|---|---|---|
| INT | 0 | | | | | |
| FM1 | 1 | Mul | f0 | f2 f4 | _ _ | No No |
| FM2 | 0 | | | | | |
| FA | 1 | Add | f6 | f8 f2 | _ _ | No No |
| FD | 1 | Div | f10 | f0 f6 | FM1 _ | No Yes |

| | I6 | I5 | | I3 |
|---|---|---|---|---|

Instruction Queue

*I1: lD f6,34(r2)*

*I2: lD f2,45(r3)*

I3: mulD f0,f2,f4

*I4: subD f8,f6,f2*

I5: divD f10,f0,f6

I6: addD f6,f8,f2

## Register result status

| | f0 | f10 | f2 | f3 | f4 | f5 | f6 | f7 | f8 |
|---|---|---|---|---|---|---|---|---|---|
| FU | FM1 | FD | | | | | FA | | |

# Example: Cycle 16

Instruction status

| Inst | Issue | ReadOp | Execute | WriteRes |
|---|---|---|---|---|
| *I1* | *1* | *2* | *3* | ***4*** |
| I2 | 5 | 6 | **7** | **8** |
| I3 | **6** | **9** | **16(r3)** | |
| I4 | **7** | **9** | **11** | **12** |
| I5 | 8 | | | |
| I6 | **13** | **14** | **16** | |

FU status

| FUID | Busy | Op | Fi | Fj Fk | Qj Qk | Rj Rk |
|---|---|---|---|---|---|---|
| INT | 0 | | | | | |
| FM1 | 1 | Mul | f0 | f2 f4 | _ _ | No No |
| FM2 | 0 | | | | | |
| FA | 1 | Add | f6 | f8 f2 | _ _ | No No |
| FD | 1 | Div | f10 | f0 f6 | FM1 _ | No Yes |

| | | | | |
|---|---|---|---|---|
| | I6 | I5 | | I3 |

Instruction Queue

*I1: lD f6,34(r2)*

*I2: lD f2,45(r3)*

**I3: mulD f0,f2,f4**

*I4: subD f8,f6,f2*

**I5: divD f10,f0,f6**

**I6: addD f6,f8,f2**

Register result status

| | f0 | f10 | f2 | f3 | f4 | f5 | f6 | f7 | f8 |
|---|---|---|---|---|---|---|---|---|---|
| FU | FM1 | FD | | | | | FA | | |

I6 finished execution, can it write the result back?

# Example: Cycle 19

Instruction status

| Inst | Issue | ReadOp | Execute | WriteRes |
|---|---|---|---|---|
| *I1* | *1* | *2* | *3* | ***4*** |
| I2 | 5 | 6 | **7** | **8** |
| I3 | **6** | **9** | **19** | |
| I4 | **7** | **9** | **11** | **12** |
| I5 | 8 | | | |
| I6 | **13** | **14** | **16** | |

FU status

| FUID | Busy | Op | Fi | Fj Fk | Qj Qk | Rj Rk |
|---|---|---|---|---|---|---|
| INT | 0 | | | | | |
| FM1 | 1 | Mul | f0 | f2 f4 | _ _ | No No |
| FM2 | 0 | | | | | |
| FA | 1 | Add | f6 | f8 f2 | _ _ | No No |
| FD | 1 | Div | f10 | f0 f6 | FM1 _ | No Yes |

| | I6 | I5 | | I3 |
|---|---|---|---|---|

Instruction Queue

*I1: lD f6,34(r2)*

*I2: lD f2,45(r3)*

I3: mulD f0,f2,f4

*I4: subD f8,f6,f2*

I5: divD f10,f0,f6

I6: addD f6,f8,f2

Register result status

| | f0 | f10 | f2 | f3 | f4 | f5 | f6 | f7 | f8 |
|---|---|---|---|---|---|---|---|---|---|
| FU | FM1 | FD | | | | | FA | | |

I6 finished execution, can not write the result back

# Example: Cycle 20

Instruction status

| Inst | Issue | ReadOp | Execute | WriteRes |
|---|---|---|---|---|
| *I1* | *1* | *2* | *3* | ***4*** |
| I2 | 5 | 6 | **7** | **8** |
| I3 | **6** | **9** | **19** | **20** |
| I4 | **7** | **9** | **11** | **12** |
| I5 | 8 | | | |
| I6 | **13** | **14** | **16** | |

FU status

| FUID | Busy | Op | Fi | Fj Fk | Qj Qk | Rj Rk |
|---|---|---|---|---|---|---|
| INT | 0 | | | | | |
| FM1 | 0 | | | | | |
| FM2 | 0 | | | | | |
| FA | 1 | Add | f6 | f8 f2 | _ _ | No No |
| FD | 1 | Div | f10 | f0 f6 | FM1 _ | Yes Yes |

| | | | | |
|---|---|---|---|---|
| | I6 | I5 | | I3 |

Instruction Queue

*I1: lD f6,34(r2)*

*I2: lD f2,45(r3)*

I3: mulD f0,f2,f4

*I4: subD f8,f6,f2*

I5: divD f10,f0,f6

I6: addD f6,f8,f2

Register result status

| | f0 | f10 | f2 | f3 | f4 | f5 | f6 | f7 | f8 |
|---|---|---|---|---|---|---|---|---|---|
| FU | FM1 | FD | | | | | FA | | |

I6 finished execution, can not write the result back

# Example: Cycle 21

Instruction status

| Inst | Issue | ReadOp | Execute | WriteRes |
|---|---|---|---|---|
| *I1* | *1* | *2* | *3* | ***4*** |
| I2 | 5 | 6 | **7** | **8** |
| I3 | **6** | **9** | **19** | **20** |
| I4 | **7** | **9** | **11** | **12** |
| I5 | 8 | **21** | | |
| I6 | **13** | **14** | **16** | |

FU status

| FUID | Busy | Op | Fi | Fj Fk | Qj Qk | Rj Rk |
|---|---|---|---|---|---|---|
| INT | 0 | | | | | |
| FM1 | 0 | | | | | |
| FM2 | 0 | | | | | |
| FA | 1 | Add | f6 | f8 f2 | _ _ | No No |
| FD | 1 | Div | f10 | f0 f6 | FM1 _ | Yes Yes |

| | | | | |
|---|---|---|---|---|
| | I6 | I5 | | |

Instruction Queue

*I1: lD f6,34(r2)*

*I2: lD f2,45(r3)*

*I3: mulD f0,f2,f4*

*I4: subD f8,f6,f2*

I5: divD f10,f0,f6

I6: addD f6,f8,f2

Register result status

| | f0 | f10 | f2 | f3 | f4 | f5 | f6 | f7 | f8 |
|---|---|---|---|---|---|---|---|---|---|
| FU | | FD | | | | | FA | | |

Now I5 can read operands (I3→I5 RAW)

# Example: Cycle 22

Instruction status

| Inst | Issue | ReadOp | Execute | WriteRes |
|---|---|---|---|---|
| I1 | 1 | 2 | 3 | 4 |
| I2 | 5 | 6 | 7 | 8 |
| I3 | 6 | 9 | 19 | 20 |
| I4 | 7 | 9 | 11 | 12 |
| I5 | 8 | 21 | 22(39) | |
| I6 | 13 | 14 | 16 | 22 |

FU status

| FUID | Busy | Op | Fi | Fj Fk | Qj Qk | Rj Rk |
|---|---|---|---|---|---|---|
| INT | 0 | | | | | |
| FM1 | 0 | | | | | |
| FM2 | 0 | | | | | |
| FA | 1 | Add | f6 | f8 f2 | _ _ | No No |
| FD | 1 | Div | f10 | f0 f6 | FM1 _ | No No |

| | | | | |
|---|---|---|---|---|
| | I6 | I5 | | |

Instruction Queue

```
I1: lD f6,34(r2)
I2: lD f2,45(r3)
I3: mulD f0,f2,f4
I4: subD f8,f6,f2
I5: divD f10,f0,f6
I6: addD f6,f8,f2
```

Register result status

| | f0 | f10 | f2 | f3 | f4 | f5 | f6 | f7 | f8 |
|---|---|---|---|---|---|---|---|---|---|
| FU | | FD | | | | | FA | | |

Now I6 can write to f6

# Example: Cycle 23

Instruction status

| Inst | Issue | ReadOp | Execute | WriteRes |
|---|---|---|---|---|
| I1 | 1 | 2 | 3 | 4 |
| I2 | 5 | 6 | 7 | 8 |
| I3 | 6 | 9 | 19 | 20 |
| I4 | 7 | 9 | 11 | 12 |
| I5 | 8 | 21 | 23(38) | |
| I6 | 13 | 14 | 16 | 22 |

FU status

| FUID | Busy | Op | Fi | Fj Fk | Qj Qk | Rj Rk |
|---|---|---|---|---|---|---|
| INT | 0 | | | | | |
| FM1 | 0 | | | | | |
| FM2 | 0 | | | | | |
| FA | 0 | | | | | |
| FD | 1 | Div | f10 | f0 f6 | FM1 _ | No No |

| | | | | |
|---|---|---|---|---|
| | | I5 | | |

Instruction Queue

```
I1: lD f6,34(r2)
I2: lD f2,45(r3)
I3: mulD f0,f2,f4
I4: subD f8,f6,f2
I5: divD f10,f0,f6
I6: addD f6,f8,f2
```

Register result status

| | f0 | f10 | f2 | f3 | f4 | f5 | f6 | f7 | f8 |
|---|---|---|---|---|---|---|---|---|---|
| FU | | FD | | | | | | | |

# Example: Cycle 61

## Instruction status

| Inst | Issue | ReadOp | Execute | WriteRes |
|---|---|---|---|---|
| I1 | 1 | 2 | 3 | 4 |
| I2 | 5 | 6 | 7 | 8 |
| I3 | 6 | 9 | 19 | 20 |
| I4 | 7 | 9 | 11 | 12 |
| I5 | 8 | 21 | 61 | |
| I6 | 13 | 14 | 16 | 22 |

## FU status

| FUID | Busy | Op | Fi | Fj Fk | Qj Qk | Rj Rk |
|---|---|---|---|---|---|---|
| INT | 0 | | | | | |
| FM1 | 0 | | | | | |
| FM2 | 0 | | | | | |
| FA | 0 | | | | | |
| FD | 1 | Div | f10 | f0 f6 | FM1 _ | No No |

| | | I5 | | |
|---|---|---|---|---|

Instruction Queue

```
I1: lD f6,34(r2)
I2: lD f2,45(r3)
I3: mulD f0,f2,f4
I4: subD f8,f6,f2
I5: divD f10,f0,f6
I6: addD f6,f8,f2
```

## Register result status

| | f0 | f10 | f2 | f3 | f4 | f5 | f6 | f7 | f8 |
|---|---|---|---|---|---|---|---|---|---|
| FU | | FD | | | | | | | |

# Example: Cycle 62

Instruction status

| Inst | Issue | ReadOp | Execute | WriteRes |
|---|---|---|---|---|
| I1 | 1 | 2 | 3 | 4 |
| I2 | 5 | 6 | 7 | 8 |
| I3 | 6 | 9 | 19 | 20 |
| I4 | 7 | 9 | 11 | 12 |
| I5 | 8 | 21 | 61 | 62 |
| I6 | 13 | 14 | 16 | 22 |

FU status

| FUID | Busy | Op | Fi | Fj Fk | Qj Qk | Rj Rk |
|---|---|---|---|---|---|---|
| INT | 0 | | | | | |
| FM1 | 0 | | | | | |
| FM2 | 0 | | | | | |
| FA | 0 | | | | | |
| FD | 0 | | | | | |

Instruction Queue

```
I1: lD f6,34(r2)
I2: lD f2,45(r3)
I3: mulD f0,f2,f4
I4: subD f8,f6,f2
I5: divD f10,f0,f6
I6: addD f6,f8,f2
```

Register result status

| | f0 | f10 | f2 | f3 | f4 | f5 | f6 | f7 | f8 |
|---|---|---|---|---|---|---|---|---|---|
| FU | | | | | | | | | |

# Example: Observations

Instruction status

| Inst | Issue | ReadOp | Execute | WriteRes |
|---|---|---|---|---|
| I1 | 1 | 2 | 3 | 4 |
| I2 | 5 | 6 | 7 | 8 |
| I3 | 6 | 9 | 19 | 20 |
| I4 | 7 | 9 | 11 | 12 |
| I5 | 8 | 21 | 61 | 62 |
| I6 | 13 | 14 | 16 | 22 |

```
I1: lD f6,34(r2)
I2: lD f2,45(r3)
I3: mulD f0,f2,f4
I4: subD f8,f6,f2
I5: divD f10,f0,f6
I6: addD f6,f8,f2
```

- Issue is in-order, execution and commit (write back) are out-of-order
- Allows instruction to be issued even if there are RAW dependence. Why this is an issue?

# Example: Observations

Instruction status

| Inst | Issue | ReadOp | Execute | WriteRes |
|---|---|---|---|---|
| I1 | 1 | 2 | 3 | 4 |
| I2 | 5 | 6 | 7 | 8 |
| I3 | 6 | 9 | 19 | 20 |
| I4 | 7 | 9 | 11 | 12 |
| I5 | 8 | 21 | 61 | 62 |
| I6 | 13 | 14 | 16 | 22 |

```
I1: lD f6,34(r2)
I2: lD f2,45(r3)
I3: mulD f0,f2,f4
I4: subD f8,f6,f2
I5: divD f10,f0,f6
I6: addD f6,f8,f2
```

- Issue is in-order, execution and commit (write back) are out-of-order
- Allows instruction to be issued even if there are RAW dependence. Why this is an issue? The FU is locked till RAW is resolved
- FU remain occupied even after execution completes, why?

# Example: Observations

Instruction status

| Inst | Issue | ReadOp | Execute | WriteRes |
|---|---|---|---|---|
| I1 | 1 | 2 | 3 | 4 |
| I2 | 5 | 6 | 7 | 8 |
| I3 | 6 | 9 | 19 | 20 |
| I4 | 7 | 9 | 11 | 12 |
| I5 | 8 | 21 | 61 | 62 |
| I6 | 13 | 14 | 16 | 22 |

```
I1: lD f6,34(r2)
I2: lD f2,45(r3)
I3: mulD f0,f2,f4
I4: subD f8,f6,f2
I5: divD f10,f0,f6
I6: addD f6,f8,f2
```

- Issue is in-order, execution and commit (write back) are out-of-order
- Allows instruction to be issued even if there are RAW dependence. Why this is an issue? The FU is locked till RAW is resolved
- FU remain occupied even after execution completes, why? To avoid WAR hazards, it can have cumulative effect of RAW + WAR
- What happens when an exception occur?

# Example: Observations

Instruction status

| Inst | Issue | ReadOp | Execute | WriteRes |
|---|---|---|---|---|
| I1 | 1 | 2 | 3 | 4 |
| I2 | 5 | 6 | 7 | 8 |
| I3 | 6 | 9 | 19 | 20 |
| I4 | 7 | 9 | 11 | 12 |
| I5 | 8 | 21 | 61 | 62 |
| I6 | 13 | 14 | 16 | 22 |

```
I1: lD f6,34(r2)
I2: lD f2,45(r3)
I3: mulD f0,f2,f4
I4: subD f8,f6,f2
I5: divD f10,f0,f6
I6: addD f6,f8,f2
```

- Issue is in-order, execution and commit (write back) are out-of-order
- Allows instruction to be issued even if there are RAW dependence. Why this is an issue? The FU is locked till RAW is resolved
- FU remain occupied even after execution completes, why? To avoid WAR hazards, it can have cumulative effect of RAW + WAR
- What happens when an exception occur? Imprecise exception, need additional OS/hw support/mechanisms

# Scoreboard: Limitations

- Amount of parallelism available in a basic block is limited
- Instruction window is dependent on # of scoreboard entries
- Execution and result write are tightly coupled ⇒ More structural hazards (impacts ILP)
- Stalls to avoid WAW and WAR ⇒ More structural hazards (impacts ILP)
- No forwarding (1 extra cycle to get the operand)
- What about memory?

# Scoreboard: Limitations

- Amount of parallelism available in a basic block is limited
- Instruction window is dependent on # of scoreboard entries
- Execution and result write are tightly coupled ⇒ More structural hazards (impacts ILP)
- Stalls to avoid WAW and WAR ⇒ More structural hazards (impacts ILP)
- No forwarding (1 extra cycle to get the operand)
- What about memory?
    - Allowing multiple memory OPs to enter the scoreboard results in added complexity in ensuring correct dependency. A simple solution is to stall LD/ST if another LD/ST is in execution.

# Dynamic scheduling: Tomasulo

- Provides dynamic scheduling capability.
- In-order issue, out-of-order execution and completion
- Supports scheduling/dispatching instructions out-of-order

| | | |
|---|---|---|
| 1 | Instruction window limited by size of basic blocks | Lays the foundation (+one step ) |
| 2 | Locking FUs before resolving RAW dependencies | Solved using reservation stations |
| 3 | Stalls to handle WAR and WAW | Register renaming through using RS |
| 4 | No support for forwarding through bypass network | Broadcast using common data bus |

# Register renaming

```
I1: add r1,r2,r4
I2: sub r5,r1,r7
I3: sw r5,100(r3)
I4: or r7,r8,r9
I5: mul r5,r8,r10
```

- What are the dependencies and possible hazards?

# Register renaming

```
I1: add r1,r2,r4
I2: sub r5,r1,r7
I3: sw r5,100(r3)
I4: or r7,r8,r9
I5: mul r5,r8,r7
```

- What are the dependencies and possible hazards?
- True dependences (RAW): `{add,sub}`, `{sub,sw}` and `{or,mul}`
- Anti-dependences (WAR): `{sub,or}` and `{sw,mul}`
- Output dependencies: `{sub,mul}`

- Register renaming can solve hazards by renaming all destination registers
- All source operands of subsequent instructions matching the renamed register renamed
- Can be effectively done with additional registers maintaining the usage status

# Register renaming

```
I1: add r1,r2,r4        I1: add r1,r2,r4
I2: sub r5,r1,r7        I2: sub r11,r1,r7
I3: sw r5,100(r3)       I3: sw r5,100(r3)
I4: or r7,r8,r9         I4: or r12,r8,r9
I5: mul r5,r8,r7        I5: mul r5,r8,r7
```

- Assume two additional registers r11 and r12
- Register  r5 and r7 are renamed to r11 and r12, respectively. WAR and WAW taken care of
- Is there any issues with the renaming?

# Register renaming (static)

```
I1: add r1,r2,r4        I1: add r1,r2,r4
I2: sub r5,r1,r7        I2: sub r11,r1,r7
I3: sw r5,100(r3)       I3: sw r11,100(r3)
I4: or r7,r8,r9         I4: or r12,r8,r9
I5: mul r5,r8,r7        I5: mul r5,r8,r12
```

- Assume two additional registers r11 and r12
- Register r5 and r7 are renamed to r11 and r12, respectively. WAR and WAW taken care of
- Is there any issues with the renaming? Yes, True dependencies must remain the same (for correctness)

# Register renaming (dynamic)

Register Alias Table (RAT)

| | |
|---|---|
| r1 | p11 |
| r2 | p32 |
| r3 | p7 |
| … | … |
| r16 | p7 |

Physical Registers

| RegID | Value |
|---|---|
| p1 | 0x123 |
| p2 | 0x0 |
| … | … |
| p32 | 0x200 |

Register Status

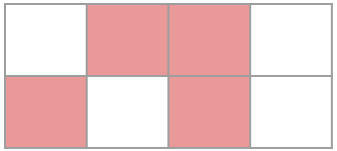
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |

- Register alias table (a.k.a rename table) is used to access the registers from the register file (physical registers)
- Register status is used to find unused registers

# Register renaming example (dynamic)

Register Alias Table (RAT)

| | |
|---|---|
| r1 | p2 |
| r2 | p3 |
| r3 | p5 |
| r4 | p7 |



Register Status

Physical Registers

| RegID | Value |
|---|---|
| p1 | 0x0 |
| p2 | 0x1000 |
| p3 | 0xAB00 |
| p4 | 0x200 |
| p5 | 0xA |
| p6 | 0x0 |
| p7 | 0x98 |
| p8 | 0x0 |

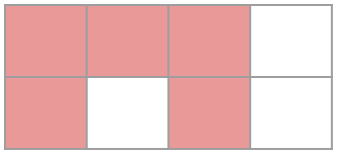
```
add r1,r2,r4
sub r2,r1,r2
mul r1,r2,r3
```

- Four architectural registers and eight physical registers
- Register allocation state is before the execution of “add” instruction

# Register renaming example (dynamic)

Register Alias Table (RAT)

| r1 | p1 |
|---|---|
| r2 | p3 |
| r3 | p5 |
| r4 | p7 |



Register Status

Physical Registers

| RegID | Value |
|---|---|
| p1 | 0x0 |
| p2 | 0x1000 |
| p3 | 0xAB00 |
| p4 | 0x200 |
| p5 | 0xA |
| p6 | 0x0 |
| p7 | 0x98 |
| p8 | 0x0 |

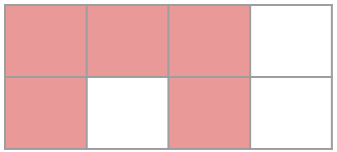
```
add r1,r2,r4    add p1,p3,p7
sub r2,r1,r2
mul r1,r2,r3
```

- Mapping of r1 is changed r1 ⟶ p2 to r1 ⟶ p1
- Free p2 ?

# Register renaming example (dynamic)

Register Alias Table (RAT)

| r1 | p1 |
|---|---|
| r2 | p3 |
| r3 | p5 |
| r4 | p7 |



Register Status

Physical Registers

| RegID | Value |
|---|---|
| p1 | 0x0 |
| p2 | 0x1000 |
| p3 | 0xAB00 |
| p4 | 0x200 |
| p5 | 0xA |
| p6 | 0x0 |
| p7 | 0x98 |
| p8 | 0x0 |

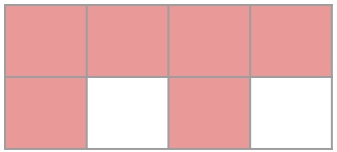
```
add r1,r2,r4    add p1,p3,p7
sub r2,r1,r2
mul r1,r2,r3
```

- Mapping of r1 is changed r1 ⟶ p2 to r1 ⟶ p1
- Free p2 ? Can not free if value of p2 is needed by a prev. instruction

# Register renaming example (dynamic)

Register Alias Table (RAT)

| | |
|---|---|
| r1 | p1 |
| **r2** | **p4** |
| r3 | p5 |
| r4 | p7 |



Register Status

Physical Registers

| RegID | Value |
|---|---|
| p1 | 0x0 |
| p2 | 0x1000 |
| p3 | 0xAB00 |
| p4 | 0x200 |
| p5 | 0xA |
| p6 | 0x0 |
| p7 | 0x98 |
| p8 | 0x0 |

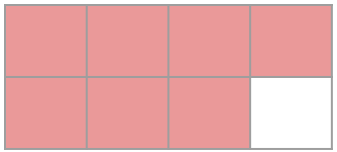
```
add r1,r2,r4    add p1,p3,p7
sub r2,r1,r2    sub p4,p1,p3
mul r1,r2,r3
```

- Mapping of r2 is changed r2 ⟶ p3 to r2 ⟶ p4
- Source operands replaced with mapping before changing the dest. mapping

# Register renaming example (dynamic)

Register Alias Table (RAT)

| r1 | p6 |
|---|---|
| r2 | p4 |
| r3 | p5 |
| r4 | p7 |



Register Status

Physical Registers

| RegID | Value |
|---|---|
| p1 | 0x0 |
| p2 | 0x1000 |
| p3 | 0xAB00 |
| p4 | 0x200 |
| p5 | 0xA |
| p6 | 0x0 |
| p7 | 0x98 |
| p8 | 0x0 |

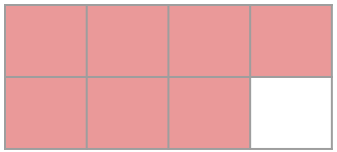
```
add r1,r2,r4    add p1,p3,p7
sub r2,r1,r2    sub p4,p1,p3
mul r1,r2,r3    mul p6,p4,p5
```

- RAW preserved, WAR and WAW eliminated
- When and how to free registers i.e., update register status?

# Register renaming example (dynamic)

Register Alias Table (RAT)

| r1 | p6 |
|---|---|
| r2 | p4 |
| r3 | p5 |
| r4 | p7 |



Register Status

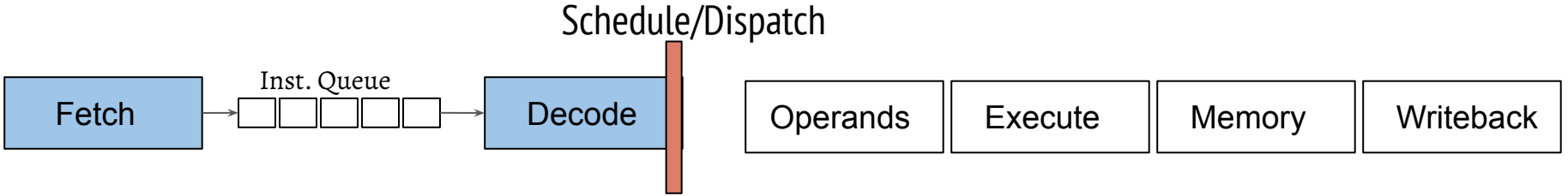
Physical Registers

| RegID | Value |
|---|---|
| p1 | 0x0 |
| p2 | 0x1000 |
| p3 | 0xAB00 |
| p4 | 0x200 |
| p5 | 0xA |
| p6 | 0x0 |
| p7 | 0x98 |
| p8 | 0x0 |

```
add r1,r2,r4   add p1,p3,p7
sub r2,r1,r2   sub p4,p1,p3
mul r1,r2,r3   mul p6,p4,p5
```

- RAW preserved, WAR and WAW eliminated
- When and how to free registers i.e., update register status?
  - Explicit tracking (using tech. like refcount)
  - Implicit freeing using reservation stations (will see)

# Dynamic Scheduling: Overall scheme

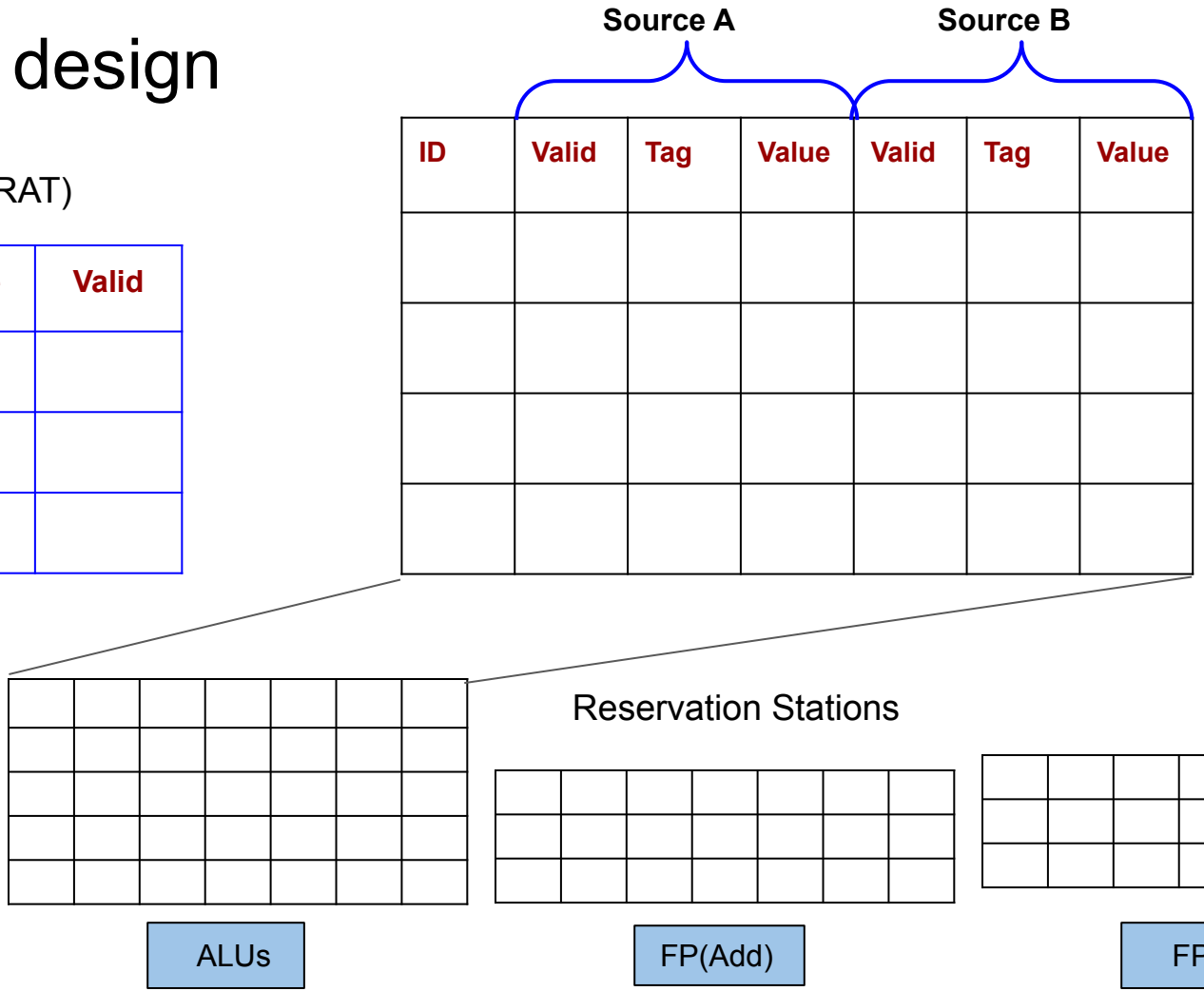



- Considering the five stage pipeline, in which stage, the dynamic scheduling should take place and why? Decode; dependency can be determined only after decoding the instructions
- What are the changes to the pipeline structure? Decode and RF access should be two different stages. Typically, an instruction queue between the fetch and decode.
- Tomasulo uses register renaming using reservation stations (RS)
- Data forwarding through a common data bus (CDB)

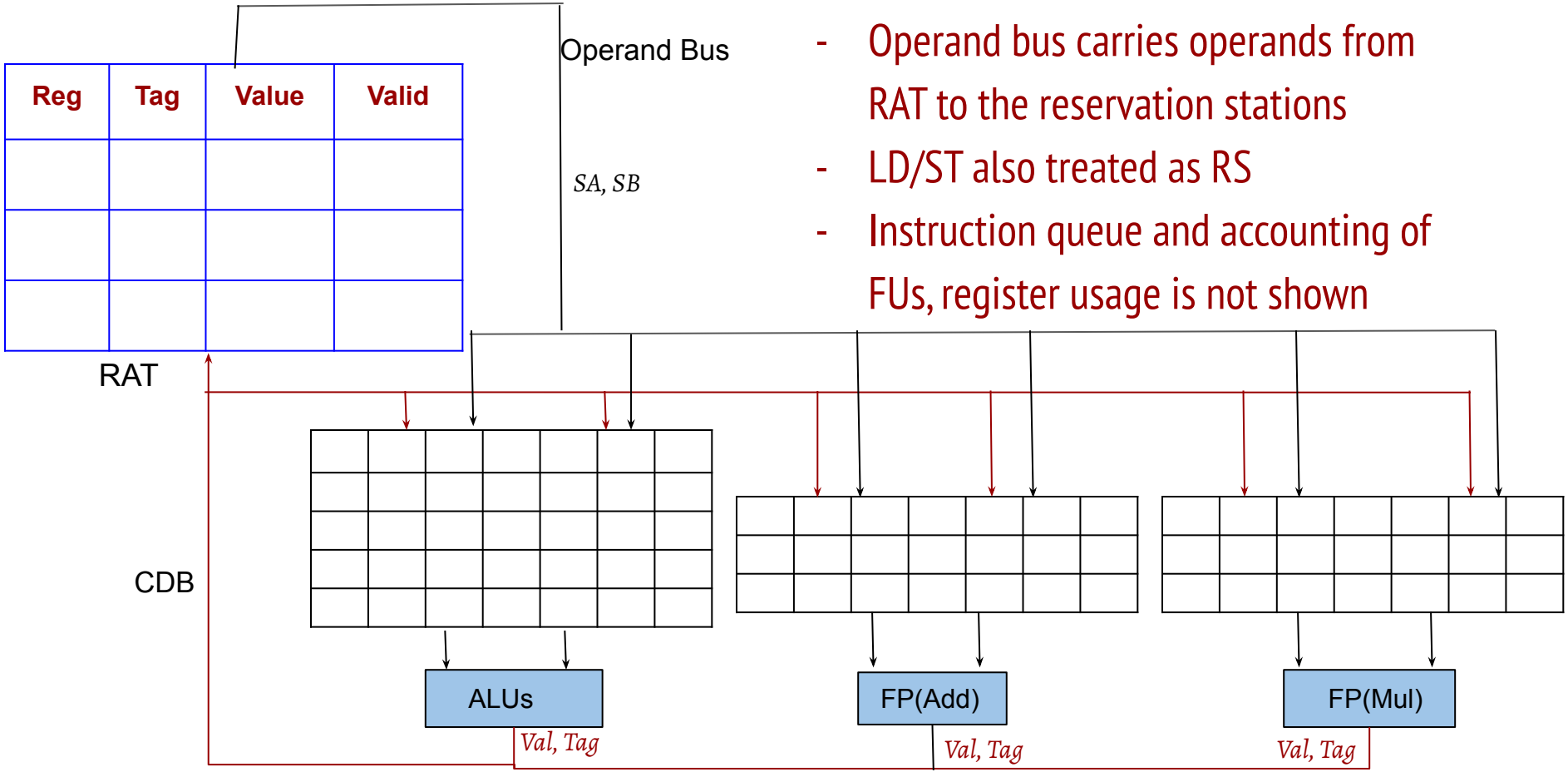
# Tomasulo's design

Register Alias Table (RAT)

| Reg | Tag | Value | Valid |
|---|---|---|---|
| | | | |
| | | | |
| | | | |

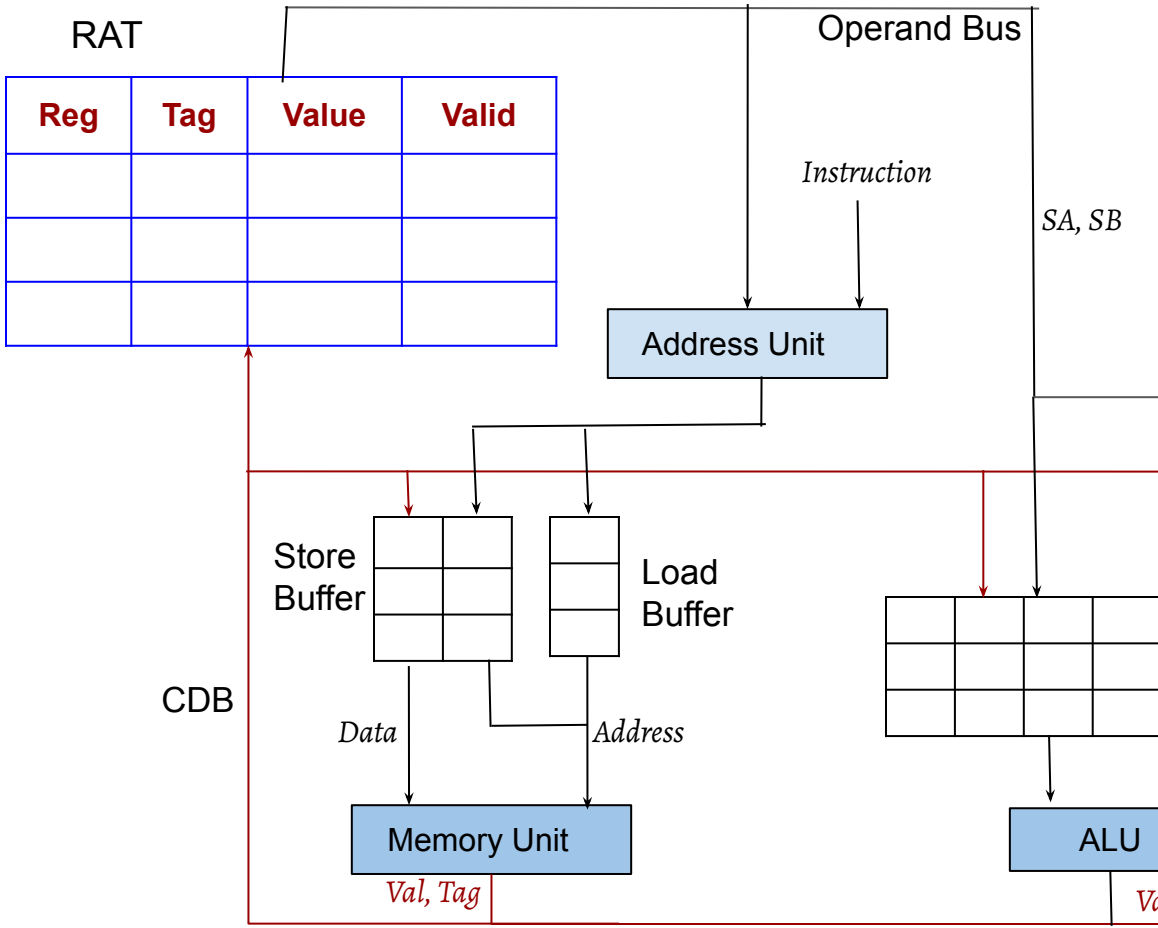


# Tomasulo's design




- Operand bus carries operands from RAT to the reservation stations
- LD/ST also treated as RS
- Instruction queue and accounting of FUs, register usage is not shown

# Tomasulo's design (with load/store)




- Load/store buffers are special reservation station
- Assumption: Address unit serializes LD/ST

# Tomasulo Algorithm

- Issue
  - Pick the instruction from the instruction queue and check for structural hazard
  - Issue the instruction to the matching RS with the source operands
  - Destination operand in the RAT is assigned the tag of reservation station

# Tomasulo Algorithm

- Issue
  - Pick the instruction from the instruction queue and check for structural hazard
  - Issue the instruction to the matching RS with the source operands
  - Destination operand in the RAT is assigned the tag of reservation station
- Execute
  - If operands are available, the FU starts execution (multiple instructions can be ready)
  - For load and store, the address unit is used to calculate the effective address before placing the load/store request in the load/store buffers

# Tomasulo Algorithm

- Issue
  - Pick the instruction from the instruction queue and check for structural hazard
  - Issue the instruction to the matching RS with the source operands
  - Destination operand in the RAT is assigned the tag of reservation station
- Execute
  - If operands are available, the FU starts execution (multiple instructions can be ready)
  - For load and store, the address unit is used to calculate the effective address before placing the load/store request in the load/store buffers
- Write result
  - When a FU finishes execution, it broadcasts the tag and value in the CDB.
  - The RAT and RS compare the tag and update the value with the broadcasted value