EEL 5764 Computer Architecture
Fall 2019

Lecture 22-23: ILP and Pipeline Scheduling

Sandip Ray
Department of Electrical and Computer Engineering
University of Florida

Announcements

- **Reminder: Mid-term 1 on October 22 8:30pm**
  - → Room NEB 0202
- EDGE Students must be registered and have appointment for the test
  - → Have Proctor U account (if you don’t have already)
  - → Schedule appointment with Proctor U to take the exam
  - → Exam title is “EEL5764- Computer Architecture – Exam 1 Fall 2019”
- No Makeup test other than for documented emergencies recognized by University of Florida
- You are allowed
  - → 1 US-letter sized crib sheet of hand-written notes (no typing). You can use both sides
  - → Scientific calculators
  - → **Nothing else is allowed during test**

Dependences

- Parallelism with **basic block** is limited
  - → Typical size of basic block = 3-6 instructions
  - → Must optimize across branches
- Increase ILP – Loop-Level Parallelism
  - → Unroll loop statically or dynamically
  - → Use SIMD (vector processors and GPUs)
- Challenges: Data dependency
  - → Instruction *j* is data dependent on instruction *i* if
    - **Instruction *i* produces a result that may be used by instruction *j***
    - **Instruction *j* is data dependent on instruction *k* and instruction *k* is data dependent on instruction *i***
- Dependent instructions cannot be executed simultaneously

Data Dependence

- Dependencies are a property of programs
- Pipeline organization determines if dependence is detected and if it causes a stall
- **Data dependence** conveys:
  - → Possibility of a hazard
  - → Order in which results must be calculated
  - → Upper bound on exploitable instruction level parallelism
- Dependencies that flow through memory locations are difficult to detect
  - → SD R1, 45(R5)
  - → LD R2, 90(R7)

Data Hazards

- A **hazard** exists between two dependent instructions that are close in a program such that overlapping their executions would change the order of accesses to the operand involved in the dependence.
- **Data Hazards**
  - → Read after write (RAW)
  - → Write after write (WAW)
  - → Write after read (WAR)

Control Dependences

- **Control Dependence** determines ordering of instruction i with respect to a branch instruction
  - → Instruction control dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch
  - → An instruction not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch

Control Dependence – Examples

- Example 1:

```
    DADDU    R1,R2,R3
    BEQZ     R4,L
    DSUBU    R1,R1,R6
L:  …
    OR       R7,R1,R8
```

- OR instruction dependent on DADDU and DSUBU
  - → Cannot move DSUBU before the branch

- Example 2:

```
    DADDU    R1,R2,R3
    BEQZ     R12,skip
    DSUBU    R4,R5,R6
    DADDU    R5,R4,R9
skip:
    OR       R7,R8,R9
```

- Reorder instruction if it does not change program semantics
- Assume R4 isn't used after skip
  - → Possible to move DSUBU before the branch

Pipeline Scheduling

- Separate dependent instruction from the source instruction by the **pipeline latency** of the source instruction
  - → Pipeline latency – cycles to separate two dependent instructions to avoid hazards

- By inserting NOP (bubbles) – pipeline stalls
- By inserting independent instructions - Scheduling
  - → Statically
  - → Dynamically

→ Scheduling relies on
- → ILP in programs
- → latencies of functional units

Static Scheduling

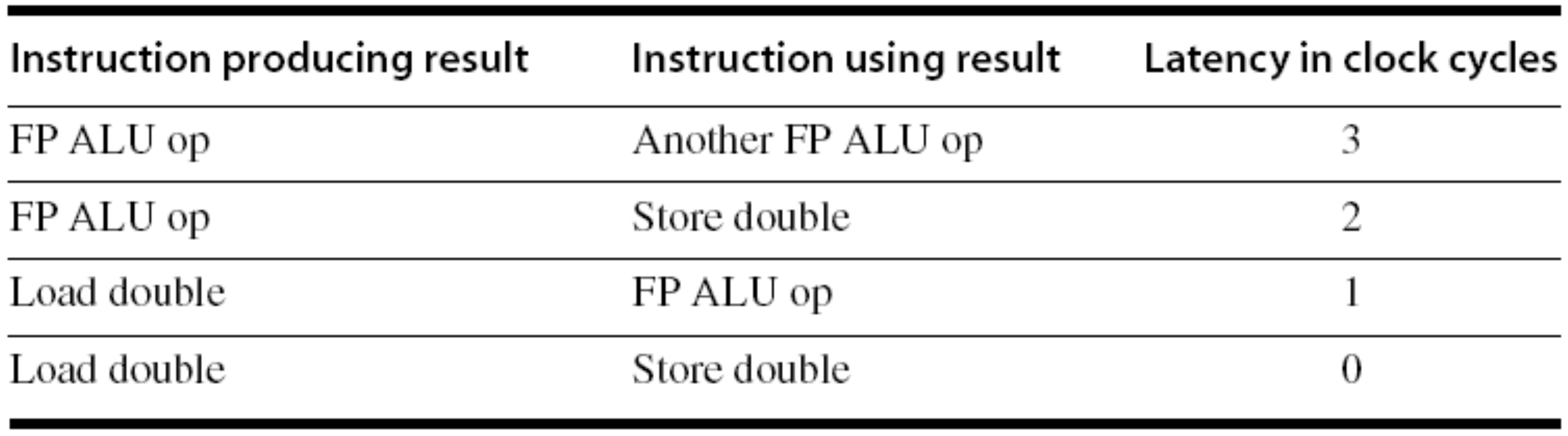
- Transform and re-arrange the code to reduce pipeline latency

- Example:

```
for (i=999; i>=0; i=i-1)
    x[i] = x[i] + s;
```

```
Loop:   L.D      F0, 0(R1)
        ADD.D    F4, F0, F2
        S.D      F4, 0(R1)
        DADDUI   R1, R1, #-8
        BNE      R1,R2,Loop
```

Assume branch delay = 1 cycle



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

Static Scheduling

```
Loop:   L.D     F0, 0(R1)
        stall
        ADD.D   F4, F0, F2
        stall
        stall
        S.D     F4, 0(R1)
        DADDUI  R1, R1, #-8
        stall
        BNE     R1,R2,Loop
```

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

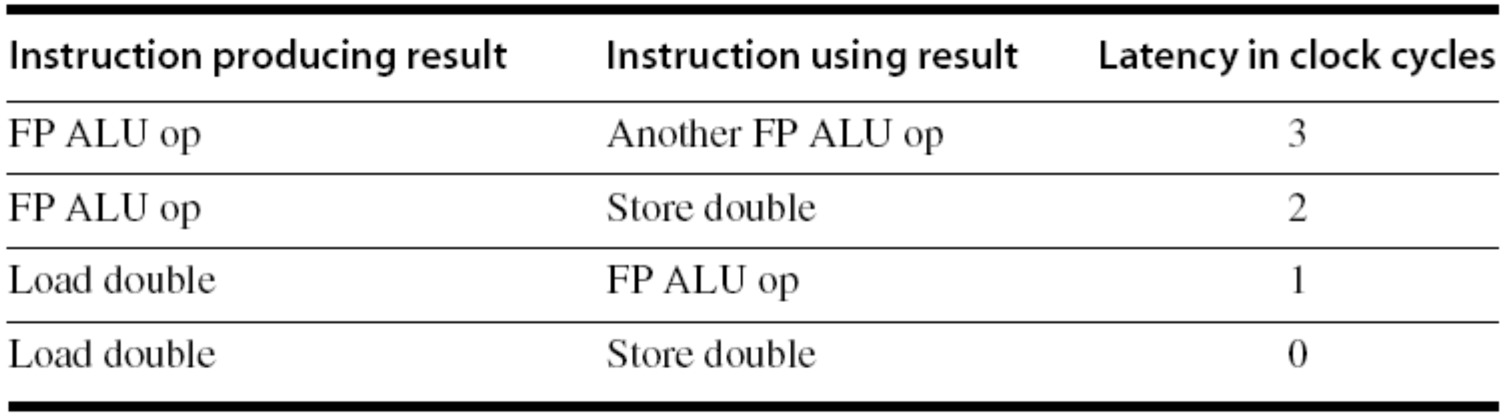
Static Scheduling

<u>Scheduled code:</u>

```
Loop:   L.D     F0, 0(R1)
        DADDUI  R1, R1, #-8
        ADD.D   F4, F0, F2
        stall
        stall
        S.D     F4, 8(R1)
        BNE     R1, R2, Loop
```

→ ILP in basic blocks is limited

→ Loop unrolling to increase ILP



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

Loop Unrolling

- Unroll by a factor of 4 (assume # elements is divisible by 4)
  → Eliminate unnecessary instructions

```
Loop:   L.D      F0, 0(R1)
        ADD.D    F4, F0, F2
        S.D      F4, 0(R1)     ;drop DADDUI & BNE
        L.D      F6, -8(R1)
        ADD.D    F8, F6, F2
        S.D      F8, -8(R1)    ;drop DADDUI & BNE
        L.D      F10, -16(R1)
        ADD.D    F12, F10, F2
        S.D      F12,- 16(R1)  ;drop DADDUI & BNE
        L.D      F14, -24(R1)
        ADD.D    F16, F14, F2
        S.D      F16, -24(R1)
        DADDUI   R1, R1, #-32
        BNE      R1, R2, Loop
```

Run in 27 cycles without scheduling

Number of live registers increase vs. original loop
→ Avoid dependence
→ Increase ILP

Loop Unrolling/Pipeline Scheduling

- Pipeline schedule the unrolled loop:

```
Loop:   L.D      F0,0(R1)
        L.D      F6,-8(R1)
        L.D      F10,-16(R1)
        L.D      F14,-24(R1)
        ADD.D    F4,F0,F2
        ADD.D    F8,F6,F2
        ADD.D    F12,F10,F2
        ADD.D    F16,F14,F2
        S.D      F4,0(R1)
        S.D      F8,-8(R1)
        DADDUI   R1, R1, #-32
        S.D      F12, -16(R1)
        S.D      F16, -24(R1)
        BNE      R1, R2, Loop
```

⇒ Run in 14 cycles after scheduling

⇒ 3.5 cycles per iteration

Dealing with Unknown Loop Bound

- Number of iterations = *n*,
- Goal: make *k* copies of the loop body
- Solution: generate pair of loops:
  - → First executes *n* mod *k* times of original loop body
  - → Second executes *n* / *k* times of unrolled loop body by *k* times.

Limitations of Loop unrolling

→ Loop iteration dependence

→ Code size increase

→ Register shortfall

Why Dynamic Scheduling

- In-Order pipeline
  - → Issue, execution, and completion of instructions follow program order
- Pipeline stalls if there is data dependence
- But, independent instructions are stalled too.

```
DIV.D          F0, F2, F4
ADD.D          F10 F0, F8
SUB.D          F12, F8, F14
```

- Static scheduling may not identify some dependences

Dynamic Scheduling

- Hardware rearrange order of instructions
  - → To reduce stalls while maintaining data flow and exception
- Advantages:
  - → Compiler doesn’t need to have knowledge of microarchitecture
  - → Handles cases where dependencies are unknown at compile time
  - → Tolerate unpredictable delays in M access, FP operations, etc.
- Disadvantage:
  - → Substantial increase in hardware complexity
  - → Complicates exceptions

Dynamic Scheduling

- ID stage is divided into
  - → Issue – decode instruction, and check for S-hazards
  - → Read Operands – wait until no D-hazards, then read operands
- Instructions pass Issue in-order
- They are stalled or bypass each other in Read Operands
  - → Enter Execute out-of-order

Dynamic Scheduling

- Dynamic scheduling implies:
  - → In-order issue
  - → Out-of-order execution
  - → Out-of-order completion

- Creates the possibility for WAR and WAW hazards

```
DIV.D       F0, F2, F4
ADD.D       F6, F0, F8
SUB.D       F8, F10, F14
MUL.D       F6, F10, F8
```

- WAR/WAW hazards can be avoided by **register renaming**

Dynamic Scheduling

- Tomasulo’s Approach
  - → Tracks when operands are available
    - ➤ To eliminate RAW hazards
  - → Introduces register renaming in hardware
    - ➤ To eliminate WAW and WAR hazards

Register Renaming

- Example:

```
DIV.D      F0,F2,F4
ADD.D      F6,F0,F8
S.D        F6,0(R1)
SUB.D      F8,F10,F14
MUL.D      F6,F10,F8
```

antidependence via F8

Output dependence via F6

Register Renaming

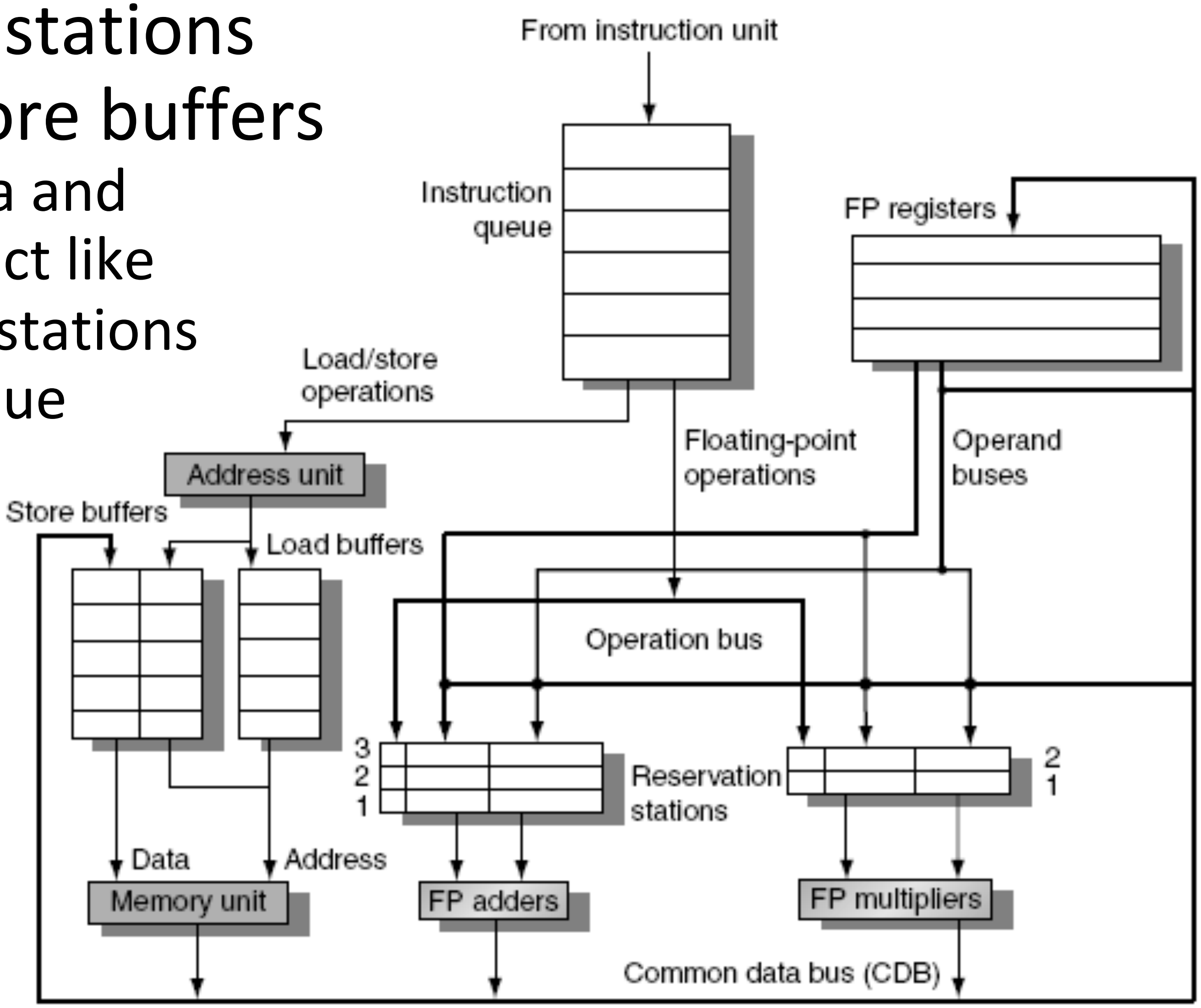
- Example: assume two additional registers S and T

```
DIV.D     F0,F2,F4
ADD.D     S, F0, F8      ; F6 -> S
S.D       S, 0(R1)       ; F6 -> S
SUB.D     T, F10, F14    ; F8 -> T
MUL.D     F6, F10, T     ; F8 -> T
```

- Now only RAW hazards remain, which must be strictly ordered

Tomasulo's Algorithm – Structure

- Reservation stations
- Load and store buffers
  - → Contain data and addresses, act like reservation stations
- → Instruction queue
- → CDB



Register Renaming By Reservation Stations

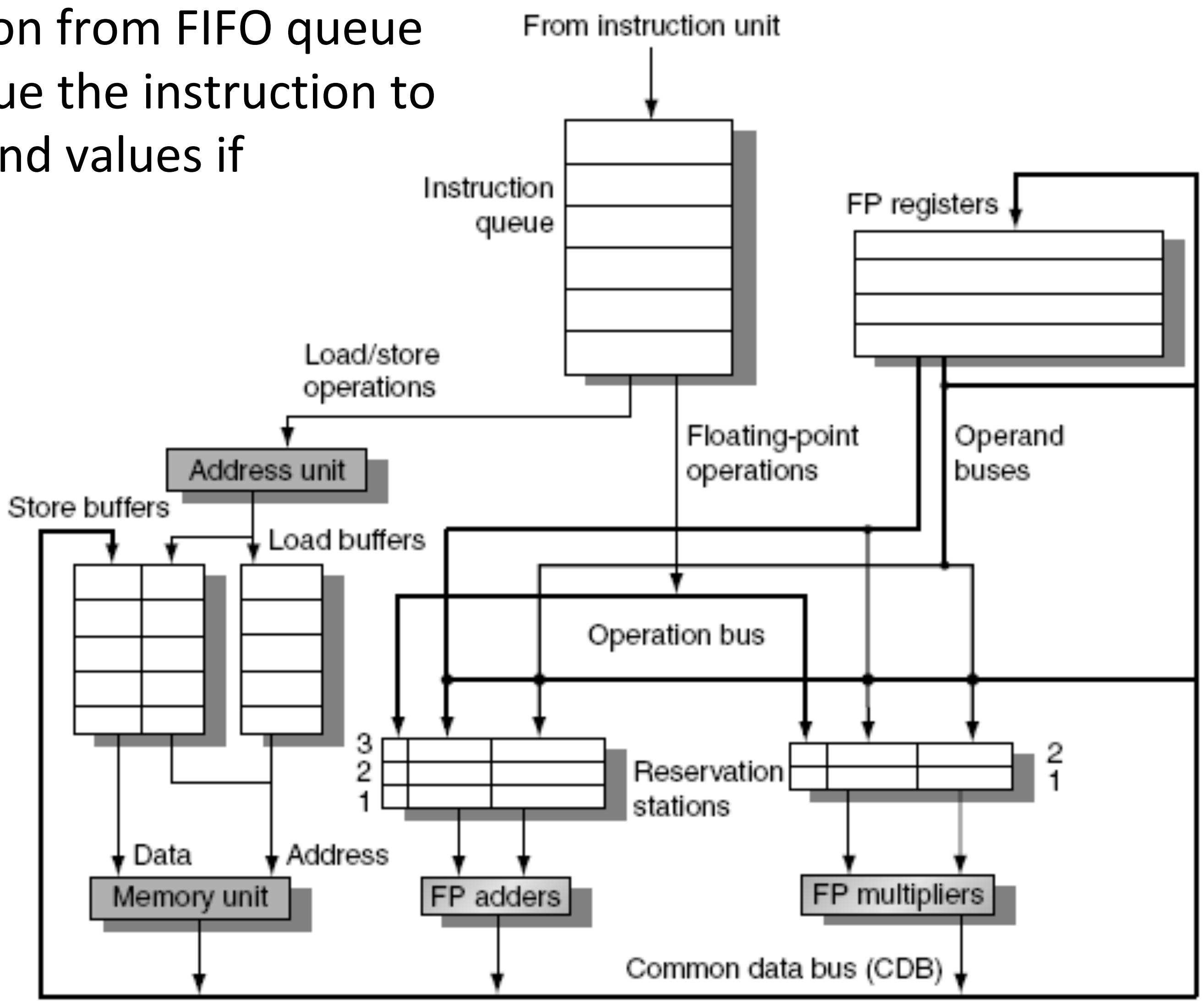
- Reservation stations (RS) buffer
  - → The instruction opcode
  - → Buffered operand values (when available)
  - → Reservation station number of instruction providing the operand values
- RS fetches and buffers an operand as soon as it becomes available (not necessarily involving register file)
- Pending instructions designate the RS to which they will send their output
- Result values broadcast on a result bus, common data bus (CDB)
- Only the last output updates the register file
- As instructions are issued, the register specifiers are renamed with the reservation station
- May be more reservation stations than registers

Tomasulo's Algorithm – Issue

- Get next instruction from FIFO queue
- If available RS, issue the instruction to the RS with operand values if available

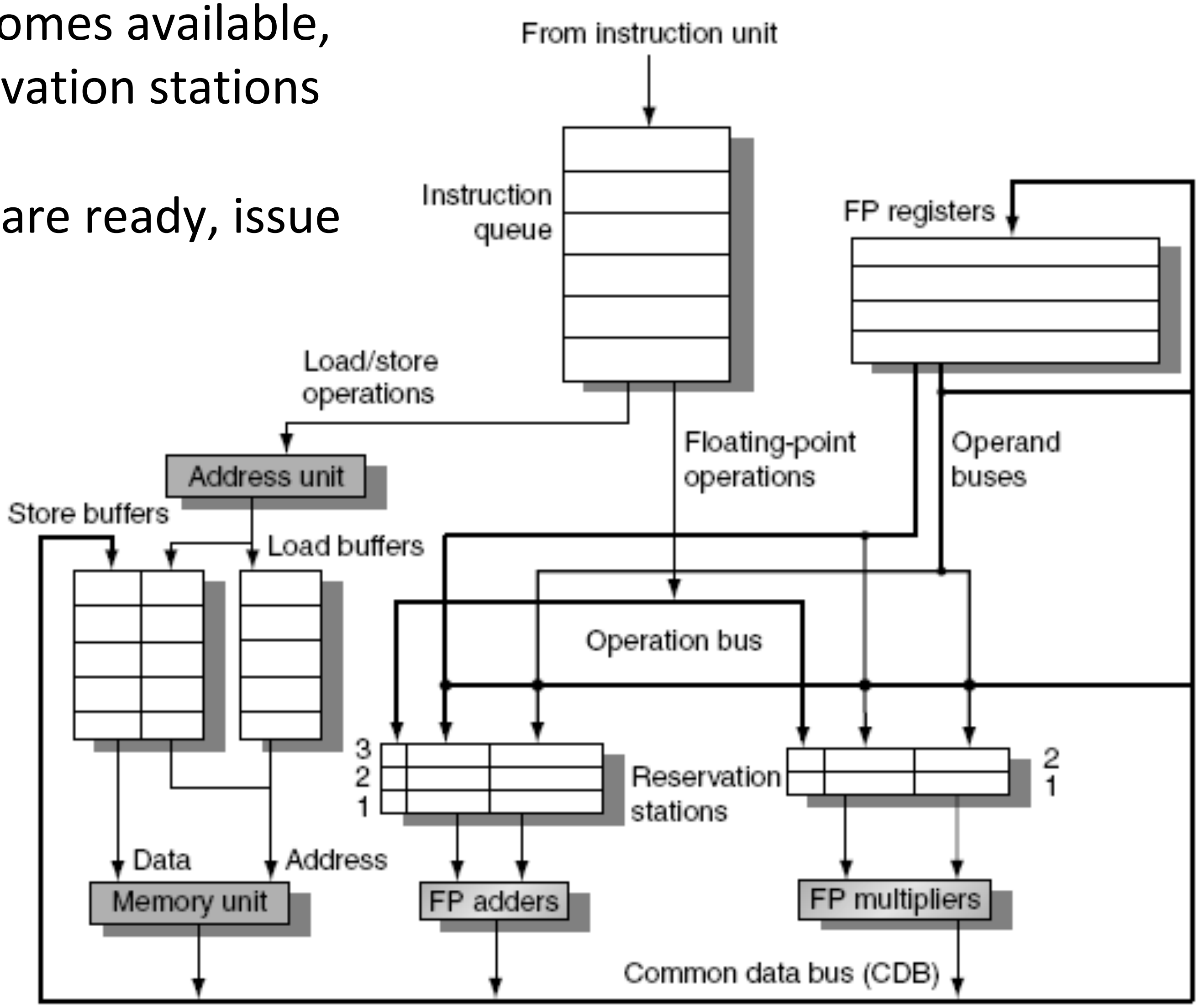
→ If operand values not available, stall the instruction

→ Renaming in this step



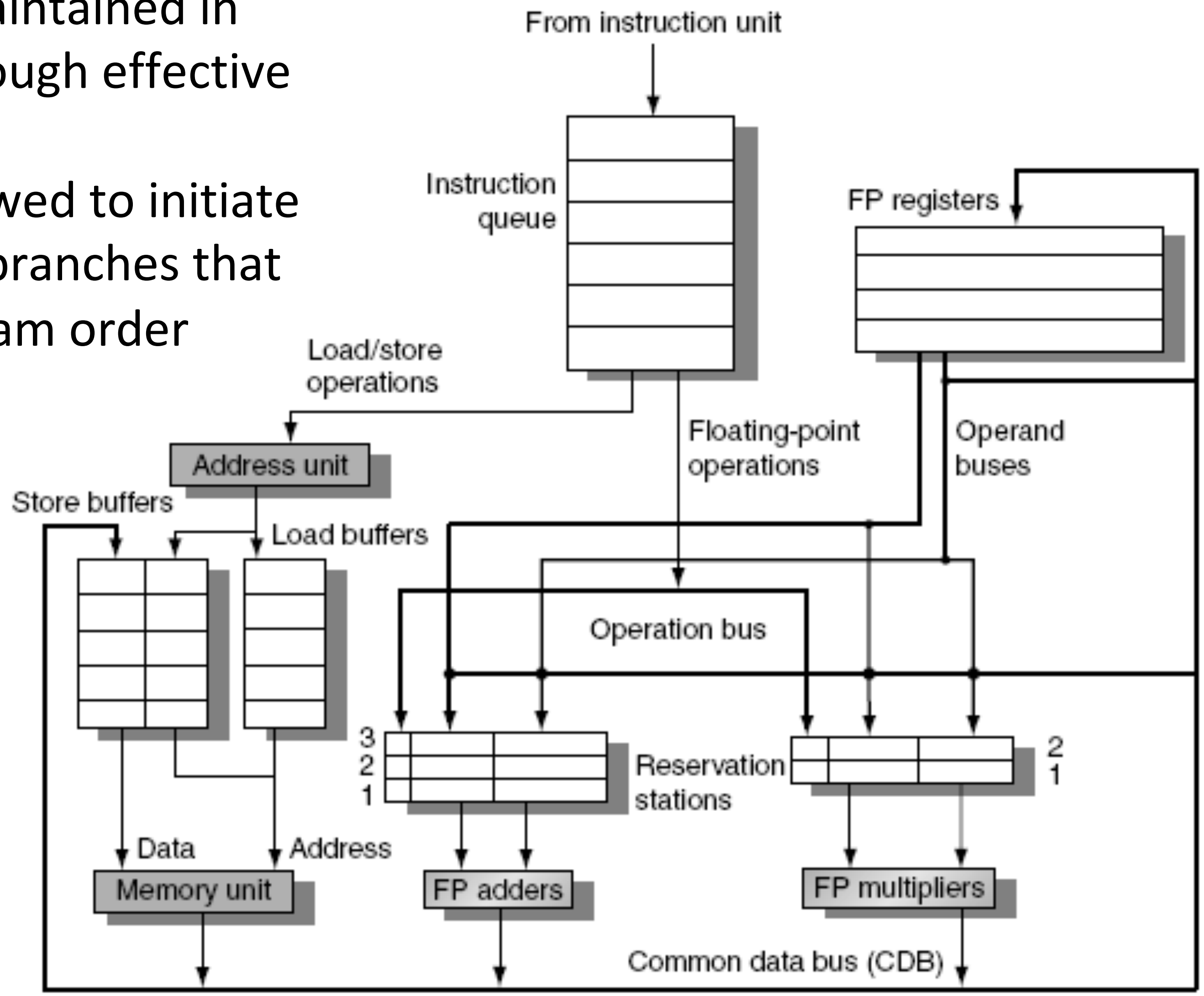
Tomasulo’s Algorithm – Execute

- When operand becomes available, store it in any reservation stations waiting for it
- When all operands are ready, issue the instruction



Tomasulo's Algorithm – Execute

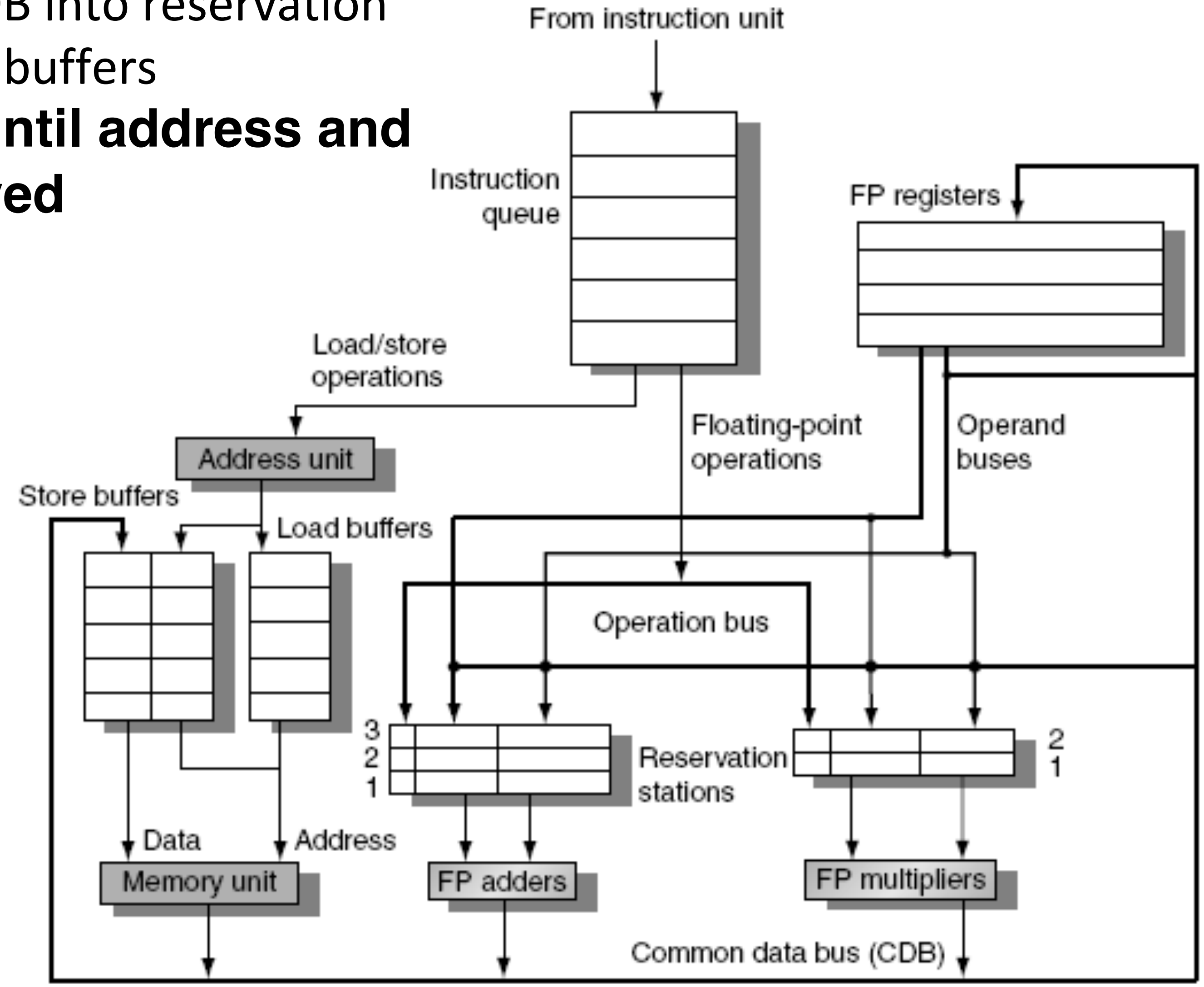
- Loads and store maintained in program order through effective address
- No instruction allowed to initiate execution until all branches that proceed it in program order have completed



Tomasulo's Algorithm – Write Result

- Write result on CDB into reservation stations and store buffers

Stores must wait until address and value are received



Tomasulo’s Algorithm – Some Notes

- Each RS, register, store buffer is attached with a tag
  - → Tags refer to FUs or buffers the produce operands
- FUs or memory put results on CDB
- RS, registers, store buffers monitor the CDB,
  - → Get the date where its tag matches tag on CDB
- Forwarding through broadcast on CDB

Example

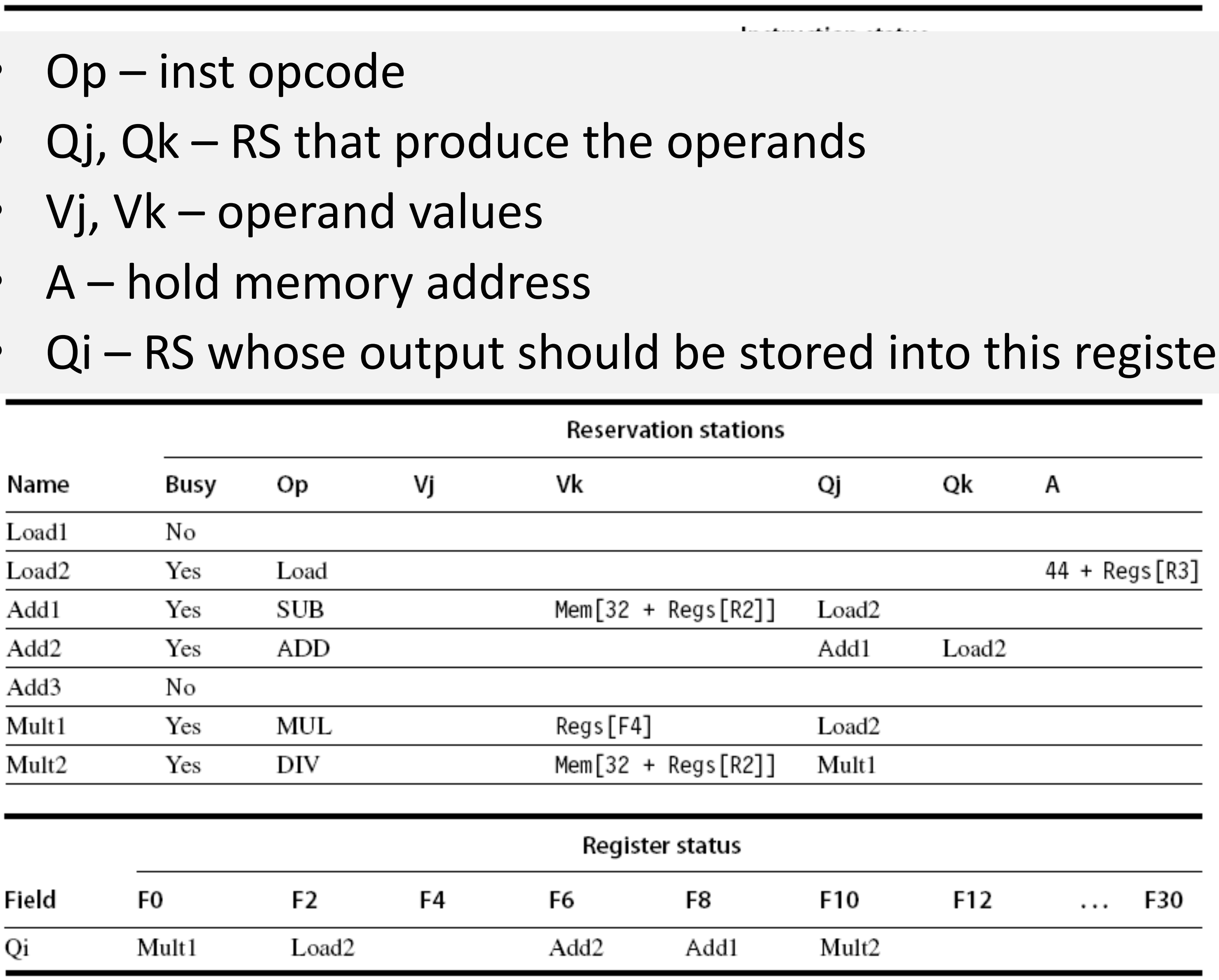
Instruction		Instruction status		
		Issue	Execute	Write Result
L.D	F6,32(R2)	√	√	√
L.D	F2,44(R3)	√	√	
MUL.D	F0,F2,F4	√		
SUB.D	F8,F2,F6	√		
DIV.D	F10,F0,F6	√		
ADD.D	F6,F8,F2	√		

	Reservation stations						
Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	No						
Load2	Yes	Load					44 + Regs[R3]
Add1	Yes	SUB		Mem[32 + Regs[R2]]	Load2		
Add2	Yes	ADD			Add1	Load2	
Add3	No						
Mult1	Yes	MUL		Regs[F4]	Load2		
Mult2	Yes	DIV		Mem[32 + Regs[R2]]	Mult1		

	Register status								
Field	F0	F2	F4	F6	F8	F10	F12	. . .	F30
Qi	Mult1	Load2		Add2	Add1	Mult2			

Reservation Station Fields

- Op – inst opcode
- Qj, Qk – RS that produce the operands
- Vj, Vk – operand values
- A – hold memory address
- Qi – RS whose output should be stored into this register



Reservation stations

Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	No						
Load2	Yes	Load					44 + Regs[R3]
Add1	Yes	SUB		Mem[32 + Regs[R2]]	Load2		
Add2	Yes	ADD			Add1	Load2	
Add3	No						
Mult1	Yes	MUL		Regs[F4]	Load2		
Mult2	Yes	DIV		Mem[32 + Regs[R2]]	Mult1		

Register status

Field	F0	F2	F4	F6	F8	F10	F12	. . .	F30
Qi	Mult1	Load2		Add2	Add1	Mult2			

A Loop Example

```
Loop:  L.D       F0, 0(R1)
       MUL.D     F4, F0, F2
       S.D       F3, 0(R1)
       DADDIU    R1, R1, -8
       BNE       R1, R2, Loop
```

Fx:	FP registers
Rx:	integer registers

Assume predict-taken for loop unrolling

		Instruction status		
Instruction	**From iteration**	**Issue**	**Execute**	**Write result**
`L.D     F0,0(R1)`	1			
`MUL.D   F4,F0,F2`	1			
`S.D     F4,0(R1)`	1			
`L.D     F0,0(R1)`	2			
`MUL.D   F4,F0,F2`	2			
`S.D     F4,0(R1)`	2			

	Reservation stations						
Name	**Busy**	**Op**	**Vj**	**Vk**	**Qj**	**Qk**	**A**
Load1							
Load2							
Add1							
Add2							
Add3							
Mult1							
Mult2							
Store1							
Store2							

	Register status							
Field	**F0**	**F2**	**F4**	**F6**	**F8**	**F10**	**F12**	**...**	**F30**
Qi									

Instruction		Instruction status			
		From iteration	Issue	Execute	Write result
`L.D`	`F0,0(R1)`	1	√	√	
`MUL.D`	`F4,F0,F2`	1	√		
`S.D`	`F4,0(R1)`	1	√		
`L.D`	`F0,0(R1)`	2	√	√	
`MUL.D`	`F4,F0,F2`	2	√		
`S.D`	`F4,0(R1)`	2	√		

	Reservation stations						
Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	Yes	Load					`Regs[R1] + 0`
Load2	Yes	Load					`Regs[R1] – 8`
Add1	No						
Add2	No						
Add3	No						
Mult1	Yes	MUL		`Regs[F2]`	Load1		
Mult2	Yes	MUL		`Regs[F2]`	Load2		
Store1	Yes	Store	`Regs[R1]`			Mult1	
Store2	Yes	Store	`Regs[R1] – 8`			Mult2	

	Register status							
Field	F0	F2	F4	F6	F8	F10	F12	...	F30
Qi	Load2		Mult2						

Ordering of Load and Store

- If load and store accesses different memory, they can be re-ordered
- Otherwise, reordering causes WAW/RAW hazards

```
L.D   F1, XXX        S.D   F2, XXX
S.D   F2, XXX        S.D   F2, YYY
```

- Consider load is next to issue
  - → Compute effective address for load
  - → Check the A field in all store buffers for match
  - → Hold load if a match is found
- Store is handled similarly
  - → Check both load and store buffers

Tomasulo's Algorithm – Summary

- Rename registers to RS
  - → To eliminates WAW and WAR hazards
- Buffer RS # for operands currently unavailable
  - → To eliminate RAW hazards
- Tolerate unpredictable delays of mem hierarchy
- Achieve high performance with compiler optimization
- HW complexity increases substantially
  - → Increase area and power