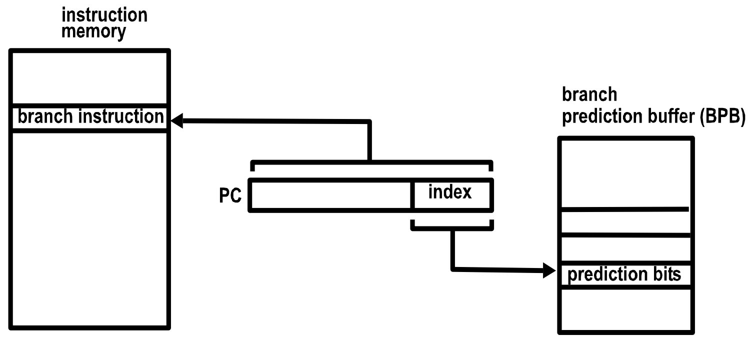
CS5375 Computer Systems Organization and Architecture

Lecture 13

Instructor: Yong Chen, Ph.D.
Department of Computer Science
Texas Tech University
Yong.Chen@ttu.edu, 806-834-0284

Review of Last Lecture

- Instruction-level parallelism (ILP) and Its Exploitation
- Control Hazards and Branch Prediction
  - Even with added hardware to compare registers, compute target and update PC in the ID stage, there is still a pipeline stall for every conditional branch (~10%) – need additional solution
- The solution is to predict the outcome of branch
  - Static branch prediction
    - Always predicts taken or not taken
  - Dynamic branch prediction
    - Hardware measures actual branch behavior (recent history) and predicts
    - Works because underlying algorithms and data being operated have regularities
    - 1-bit predictor, 2-bit predictor
    - Correlating predictors: record m most recently executed branches as taken / not taken and use that pattern to select proper n-bit branch history table, (m,n) predictor
    - Tournament predictors: dynamically choose between global and local predictors



Outline

- Exploiting ILP Using Multiple Issue and Static Scheduling
- Dynamic Scheduling

Multiple issue and Static Scheduling

- To further increase ILP
  - Multiple issue (N-way multi-issue CPU)
    - Replicate pipeline stages $\Rightarrow$ multiple pipelines (i.e., multiple washers, dryers, “folders”, “storers” in the laundry example)
    - Start multiple instructions per clock cycle
  - Launching multiple instructions per stage allows the instruction execution rate to exceed the clock rate, or in other words, CPI < 1
  - Thus it’s also often to use Instructions Per Cycle (IPC), the inverse of CPI
  - E.g., 4-way multiple-issue 4GHz CPU
    - Peak CPI = 0.25, peak IPC = 4, 16 BIPS (billion instructions per second)
  - But hazards (dependencies) reduce this in practice

Multiple Issue

- Static multiple issue (at compile time, “static”)
  - Compiler groups instructions to be issued together
  - Packages them into “issue slots”
  - Compiler detects and avoids hazards

- Dynamic multiple issue (during execution, “dynamic”)
  - CPU examines instruction stream and chooses instructions to issue each cycle
  - Compiler can help by reordering instructions
  - CPU resolves hazards using advanced techniques at runtime

Static Multiple Issue

- Compiler groups instructions into “issue packets”
  - Issue packet: a set of instructions that can be issued together in one clock cycle
    - E.g., an arithmetic instruction and a load/store instruction, if there’s no dependence/no hazards
- An issue packet can be considered as a very long instruction
  - Specifies multiple concurrent operations
    $\Rightarrow$ Very Long Instruction Word (VLIW)
  - VLIW also refers to a style of ISA that launches many operations that are defined to be independent in a single wide instruction (typically with many separate opcode fields)

RISC-V with Static Dual Issue (Two-way/Two-issue)

- Two-issue packets
  - One ALU/branch instruction
  - One load/store instruction
  - Fetching and decoding 64 bits of instructions (two instruction words)
    - ALU/branch, then load/store
    - Pad an unused instruction with nop

Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

Two-issue pipeline example (v.s. single-issue pipeline in the earlier lecture)

Hazards in the Dual-Issue RISC-V

- Ideally two-issue processor can improve performance by up to a factor of two

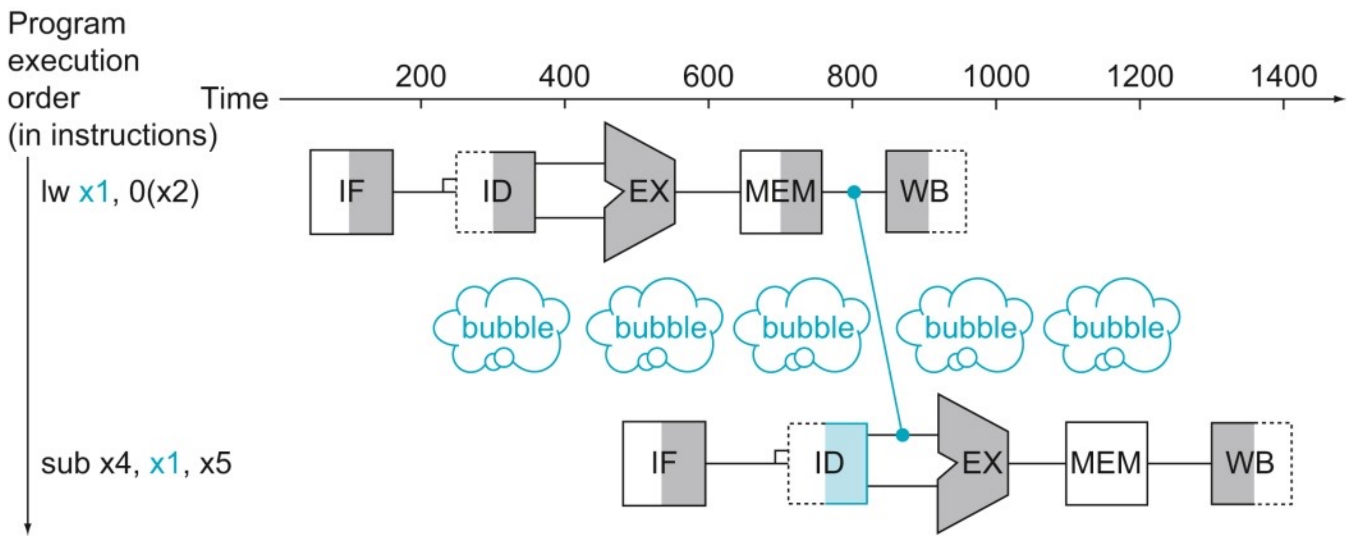
- Would require more instructions executing in parallel
  - This increases the relative performance loss from data and control hazards

- E.g., data hazard
  - Forwarding avoided stalls with single-issue
  - Now can’t use ALU result in load/store in same packet

    -
      ```
      add  x10, x0, x1
      lw   x2, 0(x10)
      ```
    - Split into two packets, effectively a stall

Hazards in the Dual-Issue RISC-V (cont.)

- Another example: load-use hazard
  - A load has a use latency of one clock cycle, which means an instruction that can use the result of the load has to be one cycle later
  - Thus, in a two-issue processor, next two instructions cannot use the load result without stalling
  - In general, the speedup is less than 2 for a two-issue processor
  - More aggressive scheduling required



Load-use data hazard

Compiler Scheduling Example

- Schedule this for dual-issue RISC-V

```
for (i=99; i>=0; i--)
   a[i] = a[i] + s;
```

```
Loop: lw    x31,0(x20)      // x31=array element
      add   x31,x31,x21     // add scalar in x21
      sw    x31,0(x20)      // store result
      addi  x20,x20,-4      // decrement pointer
      blt   x22,x20,Loop    // branch if x22 < x20
```

	ALU/branch	Load/store	cycle
Loop:	nop	lw x31,0(x20)	1
	addi x20,x20,-4	nop	2
	add x31,x31,x21	nop	3
	blt x22,x20,Loop	sw x31,4(x20)	4

CPI = 4 cycles/5 instructions = 0.8 (v.s. an ideal case of 0.5)

Or, IPC = 5/4 = 1.25 (v.s. a peak IPC = 2)

Loop Unrolling and Scheduling

- Replicate loop body to expose more parallelism
- Loop unrolling example

```
for (i=99; i>=0; i=i-4){
   a[i] = a[i] + s;
   a[i-1] = a[i-1] + s;
   a[i-2] = a[i-2] + s;
   a[i-3] = a[i-3] + s;
   }
```

Since x20 is decremented by 16 (unrolling 4 iterations), these addresses are the original value of x20 minus 4, minus 8, and minus 12

	ALU/branch	Load/store	cycle
Loop:	addi x20,x20,-16	lw x28, 0(x20)	1
	nop	lw x29, 12(x20)	2
	add x28,x28,x21	lw x30, 8(x20)	3
	add x29,x29,x21	lw x31, 4(x20)	4
	add x30,x30,x21	sw x28, 16(x20)	5
	add x31,x31,x21	sw x29, 12(x20)	6
	nop	sw x30, 8(x20)	7
	blt x22,x20,Loop	sw x31, 4(x20)	8

- IPC?

IPC = 14/8 = 1.75

Closer to 2, but at cost of registers and code size

Loop Unrolling and Scheduling (cont.)

- These sequences of instructions in this loop are actually completely independent (adding a scalar value to each element of an array)

```
Loop: lw   x31,0(x20)      // x31=array element
      add  x31,x31,x21     // add scalar in x21
      sw   x31,0(x20)      // store result
      addi x20,x20,-4      // decrement pointer
      blt  x22,x20,Loop    // branch if x22 < x20
```

- Except using x31 register in each loop iteration, which causes an enforced ordering by the reuse of a name (x31 register)
  - Store followed by a load of the same register, i.e. WAR or antidependence (a name dependence)
- Loop unrolling uses register renaming, i.e. 4 temporary registers rather than one, to resolve antidependence
  - Significant increase in code size, but delivers better performance (IPC = 1.75 or CPI = 0.57)

Outline

- Exploiting ILP Using Multiple Issue and Static Scheduling
- Dynamic Scheduling

Dynamic Scheduling

- Rearrange order of instructions to reduce stalls while maintaining data flow

- Advantages:
  - Compiler doesn’t need to have knowledge of microarchitecture to rearrange instructions
  - Handles cases where dependencies are unknown at compile time

- Disadvantages:
  - Substantial increase in hardware complexity
  - Complexity leads to the power wall

Dynamic Scheduling

- Dynamic scheduling implies:
  - Out-of-order (OOO) execution and completion

- Example 1:

```
fdiv.d f0,f2,f4
fadd.d f10,f0,f8
fsub.d f12,f8,f14
```

`fsub.d` is not dependent, can issue before `fadd.d`

Dynamic Scheduling

- Example 2:

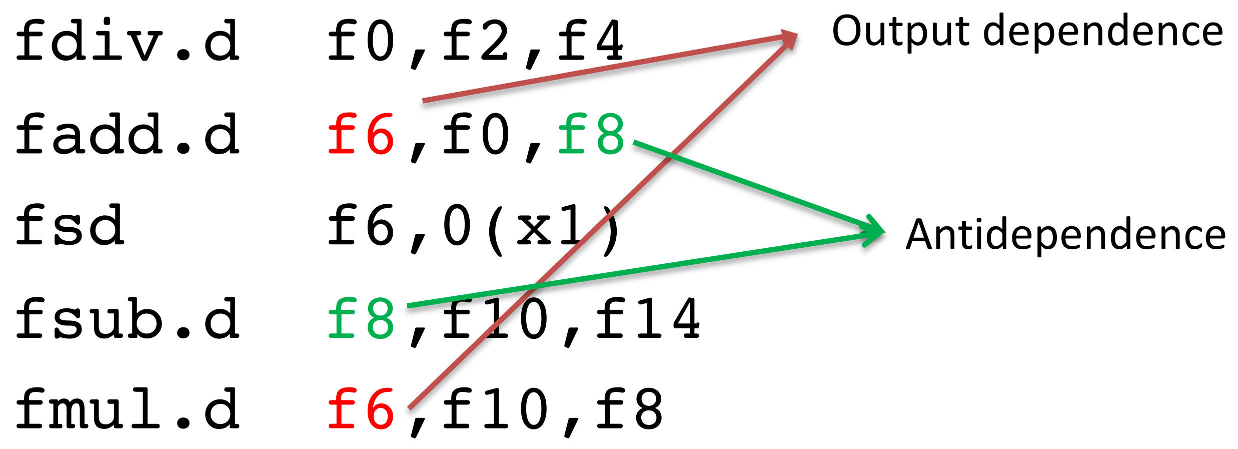
```
I1: fdiv.d f0,f2,f4
I2: fmul.d f6,f0,f8
I3: fadd.d f0,f10,f14
```

`I1 and I2:` RAW dependence (true dependence)
`I2 and I3:` WAR dependence (antidependence)
`I1 and I3:` WAW dependence (output dependence)

`fadd.d` is not dependent, but the antidependence and output dependence make it impossible to issue earlier without register renaming (e.g., use `f12` instead of `f0`, as `fadd.d f12,f10,f14`)

Register Renaming

- Example 3:



```
fdiv.d   f0,f2,f4
fadd.d   f6,f0,f8
fsd      f6,0(x1)
fsub.d   f8,f10,f14
fmul.d   f6,f10,f8
```

Register Renaming

- Example 3:

```
fdiv.d   f0,f2,f4
fadd.d   S,f0,f8
fsd      S,0(x1)
fsub.d   T,f10,f14
fmul.d   f6,f10,T
```

- Now only RAW hazards remain

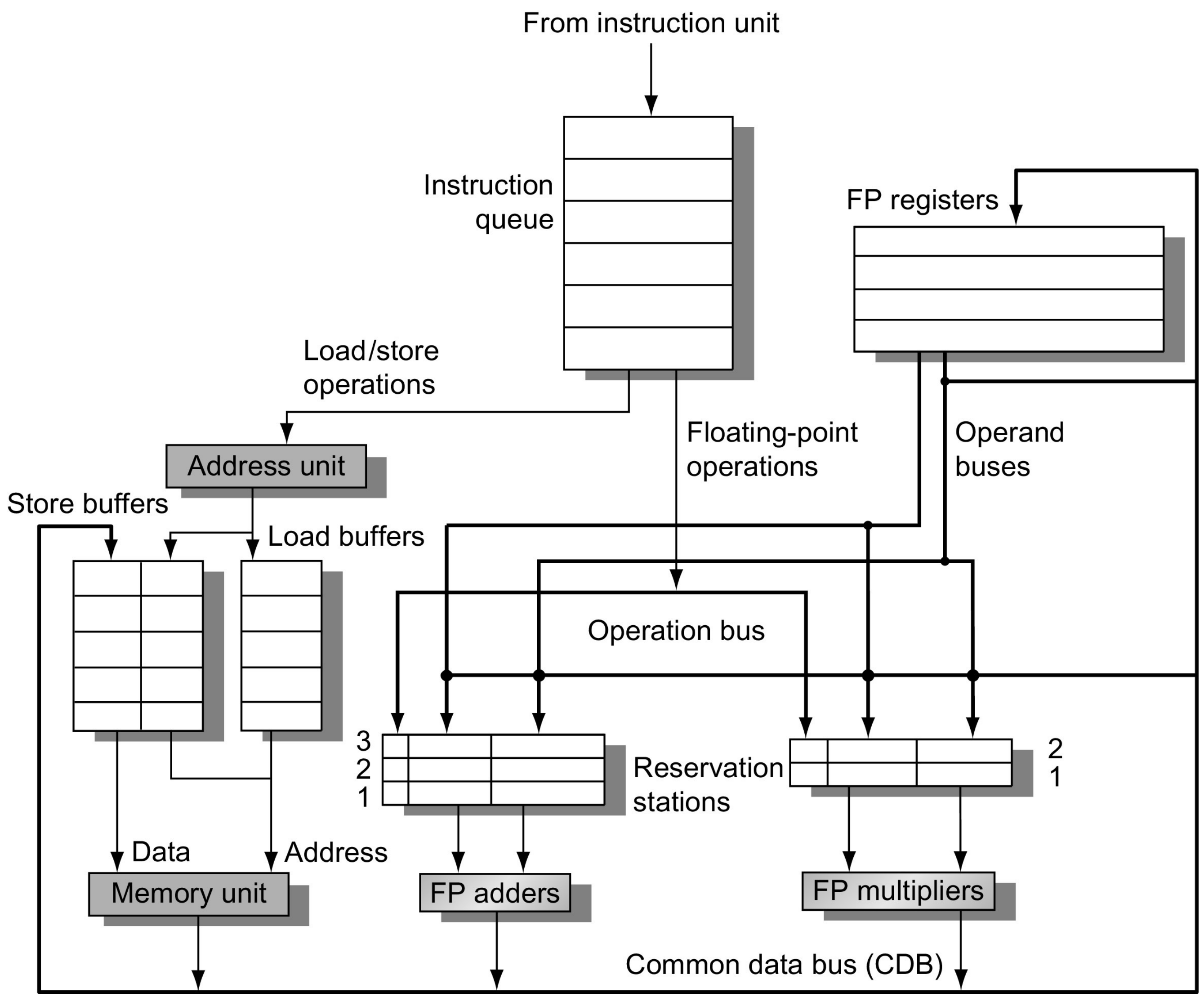
Dynamic Scheduling Using Tomasulo’s Approach

- Tomasulo’s Approach
  - A method of implementing dynamic scheduling, invented by Robert Tomasulo
  - Tracks when operands are available, satisfying the data dependence
  - Introduces register renaming in hardware to remove name dependence
    - Minimizes WAW and WAR hazards

- Register renaming is provided by reservation stations (RS)

- RS contains:
  - OP code: the operation of the instruction
  - Qj, Qk: RS producing the source operands S1 and S2
  - Vj, Vk: value of the source operands S1 and S2
  - A: the effective address for load/store instructions
  - Busy: indicates whether this RS is be used or available

Tomasulo's Algorithm



Tomasulo’s Algorithm

- Three Steps:
  - 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
  - Execute/dispatch
    - When operand becomes available, store it in any reservation stations waiting for it
    - When all operands are ready, issue the instruction
    - Loads and store maintained in program order through effective address
  - Write result
    - Write result on CDB into reservation stations and store buffers

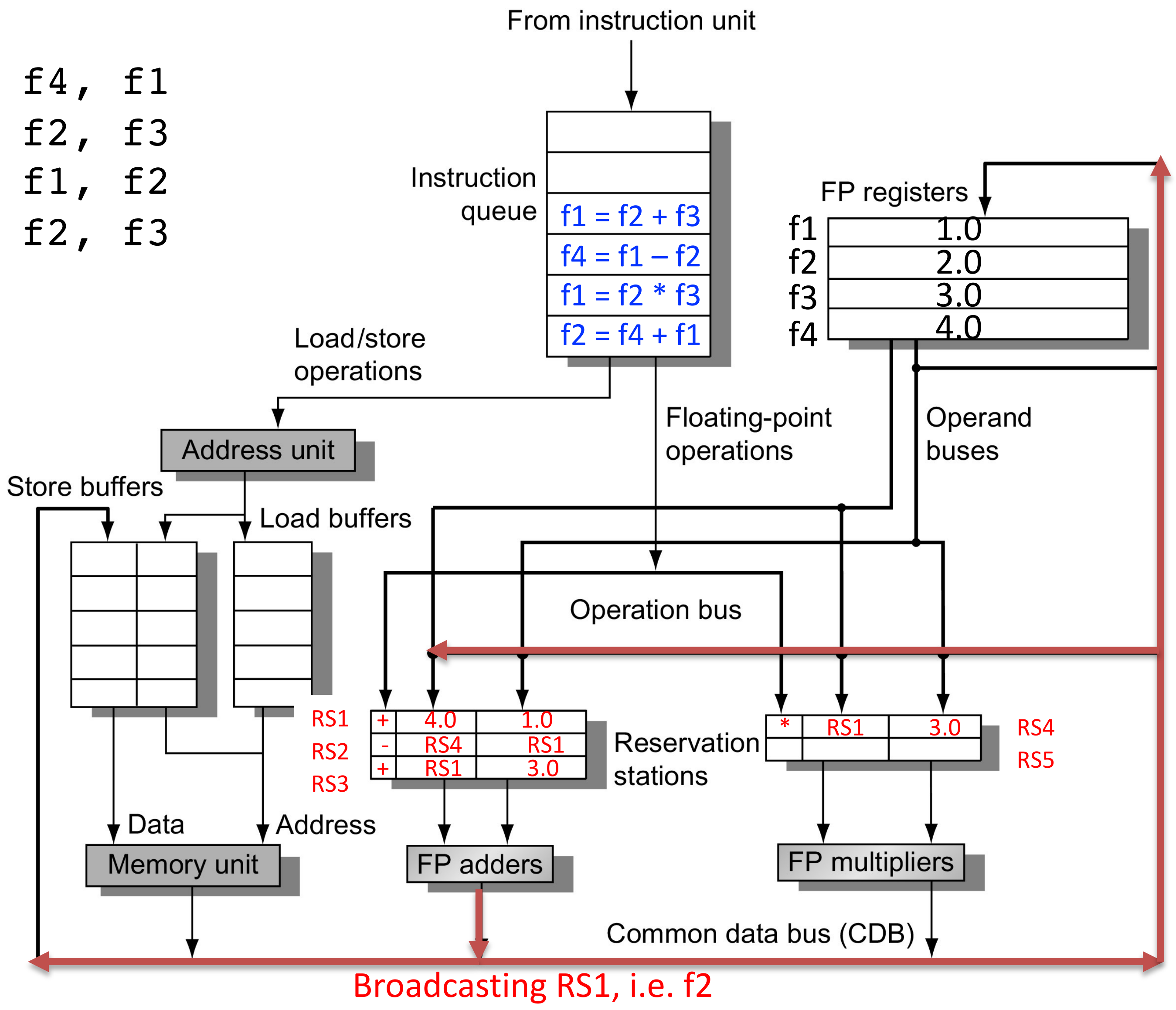
Tomasulo's Algorithm: An Example

```
I1: fadd.d f2, f4, f1
I2: fmul.d f1, f2, f3
I3: fsub.d f4, f1, f2
I4: fadd.d f1, f2, f3
```

Register alias table

f1	~~RS4~~RS3
f2	RS1
f3	
f4	RS2

If there's no RS3, we cannot issue `I4`.



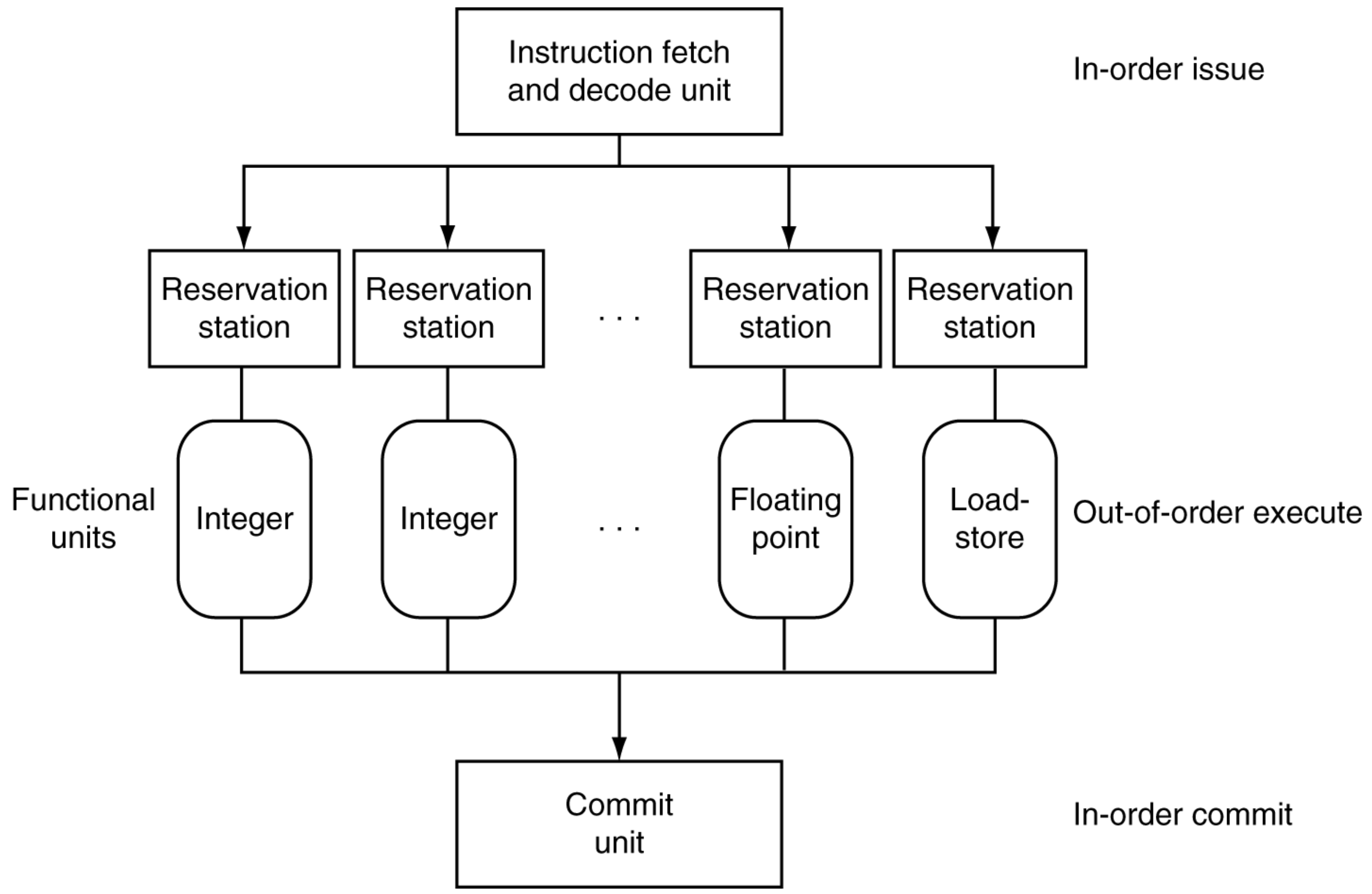
Such a process continues as issuing "in order" but executing "out of order".

CPUs nowadays use similar mechanism to issue all instructions with 100s instructions in IQ.

Dynamic Multiple Issue

- Also known as “superscalar” processors, or “dynamic pipeline scheduling”
- CPU decides whether to issue 0, 1, 2, or more instructions each cycle
  - Avoiding structural and data hazards
- Avoids the need for compiler scheduling

Dynamic Multiple Issue CPU



Dynamic Scheduling, Multiple Issue, and Speculation

- Modern microarchitectures:
  - Dynamic scheduling + multiple issue + speculation
- How much to speculate
  - Mis-speculation degrades performance and power relative to no speculation
    - May cause additional misses (cache, TLB)
  - Prevent speculative code from causing higher costing misses (e.g. L2)
- Speculating through multiple branches
  - Complicates speculation recovery
- Speculation and energy efficiency
  - Note: speculation is only energy efficient when it significantly improves performance

Readings

- Chapter 3, 3.4 – 3.8