Chapter 3 – Instruction-Level Parallelism and its Exploitation (Part 1)

ILP vs. Parallel Computers

Dynamic Scheduling (Section 3.4, 3.5)

Dynamic Branch Prediction (Section 3.3, 3.9, and Appendix C)

Hardware Speculation and Precise Interrupts (Section 3.6)

Multiple Issue (Section 3.7)

Static Techniques (Section 3.2, Appendix H)

Limitations of ILP

Multithreading (Section 3.11)

Putting it Together (Mini-projects)

ILP vs. Parallel Computers

Instruction-Level Parallelism (ILP)

- Instructions of single process (or thread) executed in parallel
- Parallel components must *appear* to execute in sequential program order

Parallel Computers or Multiprocessors

- Program divided into multiple processes (or threads)
- Instructions of multiple threads executed in parallel
- Typically also involves ILP within each thread
- No a priori sequential order between parallel threads

Dynamic Scheduling - Basics

The situation:

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

The problem:

ADD stalls due to RAW hazard

MULT stalls because ADD stalls

Example

```
        1   2   3   4   5   6   7   8
DIV.D  IF  ID  E/  E/  E/  E/  MEM WB
ADD.D      IF  ID  **  **  **  E+  E+
MULT.D         IF  **  **  **  ID  E* why stall?
```

In-order execution limits performance

Dynamic Scheduling - Basics (Cont.)

Solutions

- Static Scheduling
- Dynamic Scheduling

Static Scheduling (Software)

- Compiler reorganizes instructions
- +
- +
- (Will see more later)

Dynamic Scheduling (Hardware)

- Hardware reorganizes instructions
- +
- +

Dynamic Scheduling - Basics (Cont.)**

Solutions

- Static Scheduling
- Dynamic Scheduling

Static Scheduling (Software)

- Compiler reorganizes instructions
- + Simpler hardware
- +
- (Will see more later)

Dynamic Scheduling (Hardware)

- Hardware reorganizes instructions
- +
- +

*Dynamic Scheduling - Basics (Cont.)***

Solutions

- Static Scheduling
- Dynamic Scheduling

Static Scheduling (Software)

- Compiler reorganizes instructions
- + Simpler hardware
- + Can use more powerful algorithms
- (Will see more later)

Dynamic Scheduling (Hardware)

- Hardware reorganizes instructions
- +
- +

*Dynamic Scheduling - Basics (Cont.)***

Solutions

- Static Scheduling
- Dynamic Scheduling

Static Scheduling (Software)

- Compiler reorganizes instructions
- + Simpler hardware
- + Can use more powerful algorithms
- (Will see more later)

Dynamic Scheduling (Hardware)

- Hardware reorganizes instructions
- + Handles dependences unknown at compile time
- +

Dynamic Scheduling - Basics (Cont.)**

Solutions

- Static Scheduling
- Dynamic Scheduling

Static Scheduling (Software)

- Compiler reorganizes instructions
- + Simpler hardware
- + Can use more powerful algorithms
- (Will see more later)

Dynamic Scheduling (Hardware)

- Hardware reorganizes instructions
- + Handles dependences unknown at compile time
- + Software is more portable

Dynamic Scheduling - Basics (Cont.)

- In-order execution - Static
  - Instructions sent to execution units sequentially
  - Stall instruction $i + 1$ if instruction i stalls for lack of operands
- Out-of-order execution - Dynamic
  - Send independent instructions to execution units as soon as possible

Dynamic Scheduling Basics (Cont.)

Original simple pipeline

- ID – decode, check all hazards, read operands
- EX – execute

Dynamic pipeline

- Split ID ("issue to execution unit") into two parts
  - Check for structural hazards
  - Wait for data dependences
- New organization (conceptual):
  - Issue – decode, check structural hazards, read ready operands
  - ReadOps – wait until data hazards clear, read operands, begin execution
  - *Issue stays in-order; ReadOps/beginning of EX is out-of-order*

*Dynamic Scheduling Basics (Cont.)***

Original simple pipeline

- ID – decode, check all hazards, read operands
- EX – execute

Dynamic pipeline

- Split ID (“issue to execution unit”) into two parts
  - Check for structural hazards
  - Wait for data dependences
- New organization (conceptual):
  - Issue – decode, check structural hazards, read ready operands
  - ReadOps – wait until data hazards clear, read operands, begin execution

Issue

Dispatch *~~Issue~~ stays in-order; ~~ReadOps~~/beginning of EX is out-of-order*

Dynamic Scheduling Basics (Cont.)

Dynamic scheduling can create WAW, WAR hazards, and imprecise exceptions

WAW hazards with dynamic scheduling

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

WAR hazards with dynamic scheduling

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

Can always stall,

but more aggressive solution with ***register renaming***

Register Renaming - Tomasulo’s Algorithm

Registers are *Names* for data values

- Think of register specifiers as *tags*
- NOT storage locations
- *Tomasulo's algorithm exploited above in IBM 360/91*

WAW hazards:

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

WAR hazards:

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

Some History - IBM 360/91

Fast 360 for scientific code

- Completed in 1967
- Predates cache memories

Pipelined, rather than multiple, functional units (FU)

- We will assume multiple functional units

360 had register memory instructions, we don’t

Register Renaming - Tomasulo's Algorithm

Tomasulo's algm uses *reservation stations* for register renaming

Instruction is "issued" to a reservation station

A pending operand is designated via a tag

- Tag = reservation station that will provide the operand

Reservation station with pending instruction fetches and buffers the operand when it becomes available

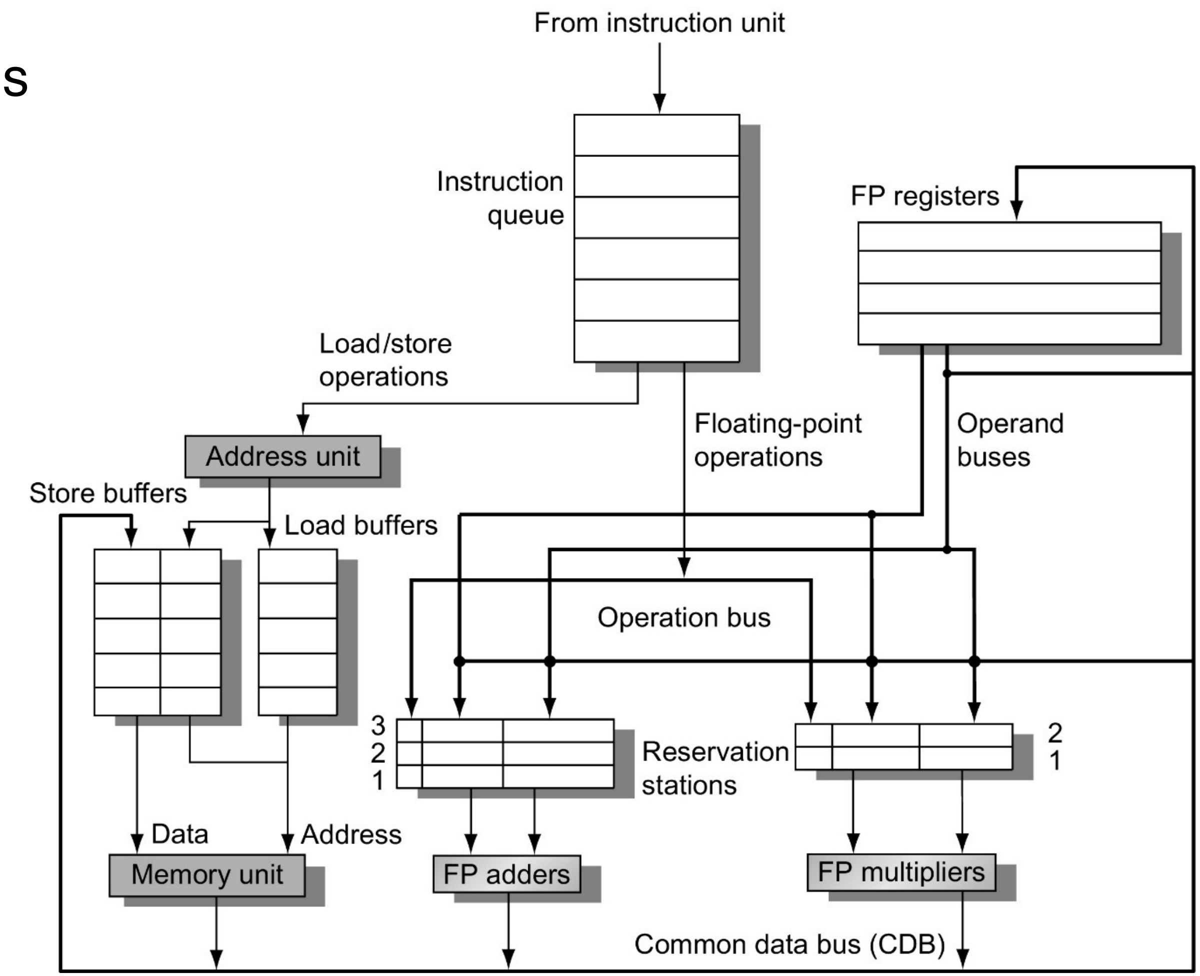
All FUs place output on the *common data bus* (CDB) with tag

Waiting reservation station gets the data from the CDB (register bypass)

Tomasulo's Algorithm - Implementation

Extend simple pipeline as example for Tomasulo's algorithm

- Assume multiple FUs



Tomasulo's Algorithm – Implementation**

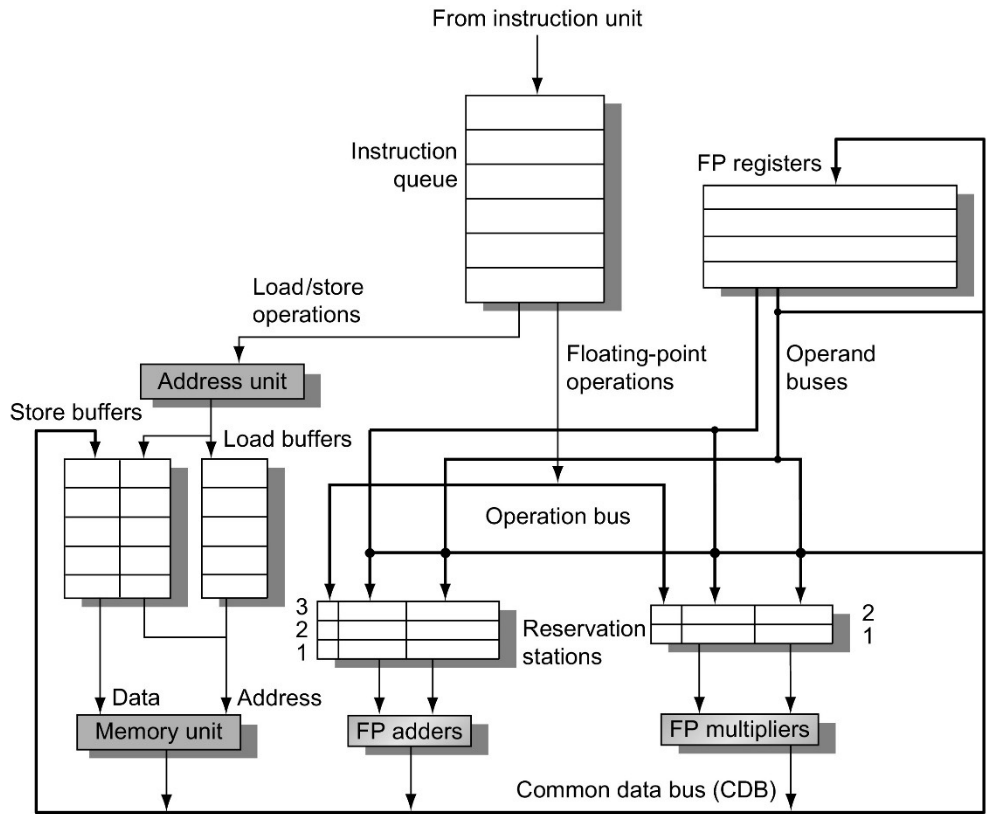


Figure 3.10 The basic structure of a RISC-V floating-point unit using Tomasulo's algorithm. Instructions are sent from the instruction unit into the instruction queue from which they are issued in first-in, first-out (FIFO) order. The reservation stations include the operation and the actual operands, as well as information used for detecting and resolving hazards. Load buffers have three functions: (1) hold the components of the effective address until it is computed, (2) track outstanding loads that are waiting on the memory, and (3) hold the results of completed loads that are waiting for the CDB. Similarly, store buffers have three functions: (1) hold the components of the effective address until it is computed, (2) hold the destination memory addresses of outstanding stores that are waiting for the data value to store, and (3) hold the address and value to store until the memory unit is available. All results from either the FP units or the load unit are put on the CDB, which goes to the FP register file as well as to the reservation stations and store buffers. The FP adders implement addition and subtraction, and the FP multipliers do multiplication and division.

Our Tomasulo Pipeline

3-stage Execution (ignore IF and MEM)

Issue	Get instruction from queue ALU Op: Check for available reservation station Load/Store: Check for available load/store buffer If not, stall due to structural hazard
Execute	If operands available, execute operation If not, monitor CDB for operand
Write	If CDB available, write it on CDB If not, stall

Our Tomasulo Pipeline, cont

Reservation Stations

- Handle distributed hazard detection and instruction control

Everything, except store buffers, has a *tag*

- 4-bit tag specifies reservation station or load buffer
- Specifies which FU will produce result

Register specifier is used to assign tags

- THEN IT'S DISCARDED!
- Register specifers are ONLY used in ISSUE

Our Tomasulo Pipeline, cont

Reservation Stations

Op	Opcode
Q_j, Q_k	Tag Fields
V_j, V_k	Operand values
Busy	Currently in use

Register File and Store Buffer

Q_i	Tag Field
Busy	Currently in use

Load and Store Buffers

Busy	Currently in use
A	Address

Latencies: FP+ = 2, FP* = 10, FP/ = 40, Load/int = 1

Tomasulo Example

Example code

```
L.D     F6,34(R2)

L.D     F2,45(R3)

MULT.D  F0,F2,F4

SUB.D   F8,F6,F4

DIV.D   F10,F0,F6

ADD.D   F6,F8,F2
```

Tomasulo Example

Instruction Status (For illustration ONLY)				
Instruction		Issue	Execute	Write
L.D	F6,34(R2)			
L.D	F2,45(R3)			
MULT.D	F0,F2,F4			
SUB.D	F8,F6,F4			
DIV.D	F10,F0,F6			
ADD.D	F6,F8,F2			

FU	Name	Busy	Op	Vj	Vk	Qj	Qk
1	Add1						
2	Add2						
3	Add3						
4	Mult1						
5	Mult2						

Register Result Status									
	F0	F2	F4	F6	F8	F10	F12	…	F30
QI									
Busy									

Tomasulo Example

Instruction Status (For illustration ONLY)				
Instruction		Issue	Execute	Write
L.D	F6,34(R2)			
L.D	F2,45(R3)			
MULT.D	F0,F2,F4			
SUB.D	F8,F6,F4			
DIV.D	F10,F0,F6			
ADD.D	F6,F8,F2			

FU	Name	Busy	Op	Vj	Vk	Qj	Qk
1	Add1						
2	Add2						
3	Add3						
4	Mult1						
5	Mult2						

Register Result Status									
	F0	F2	F4	F6	F8	F10	F12	…	F30
QI									
Busy									

Tomasulo Example

Instruction Status (For illustration ONLY)				
Instruction		Issue	Execute	Write
L.D	F6,34(R2)			
L.D	F2,45(R3)			
MULT.D	F0,F2,F4			
SUB.D	F8,F6,F4			
DIV.D	F10,F0,F6			
ADD.D	F6,F8,F2			

FU	Name	Busy	Op	Vj	Vk	Qj	Qk
1	Add1						
2	Add2						
3	Add3						
4	Mult1						
5	Mult2						

Register Result Status									
	F0	F2	F4	F6	F8	F10	F12	…	F30
QI									
Busy									

*Variations on Tomasulo Example***

What if the last ADD is replaced with ADD.D ***F10***, F8, F2?

What if we add another instruction at the end: ADD.D F6, F0, F2?

Tomasulo, cont.

Out-of-order loads and stores?

CDB is a bottleneck

- Could duplicate
- Increases the required hardware

Complex implementation

*Tomasulo, cont***

- Out-of-order loads and stores?
  - What about WAW, RAW, and WAR hazards?
    - Compare all load addresses w/ address in store buffers
    - Compare all store addresses w/ address in load/store buffers
      - Stall if they match
- CDB is a bottleneck
  - Could duplicate
  - Increases the required hardware
- Complex implementation

Tomasulo, cont.

Advantages

- Distribution of hazard detection
- Elimination of WAR and WAW stalls

Common Data Bus

- \+ Broadcasts results to multiple instructions, bypasses registers
- \- Central bottleneck
  - Could duplicate (increases required hardware)

Register Renaming

- \+ Eliminates WAR and WAW Hazards
- \+ Allows dynamic loop unrolling
  - Especially important with only 4 registers
- \- Requires many associative lookups

Loops with Tomasulo's Algorithm

Consider the following example:

```
FORTRAN:
DO I = 1, N
     C[I] = A[I] + s * B[I]

ASSEMBLY:
L.D    F0, A(R1)
L.D    F2, B(R1)
MUL.D  F2, F2, F4 /* s in F4 */
ADD.D F2, F2, F0
S.D    C(R1), F2
Branch code
```

What would Tomasulo's algorithm do?