CS60003: High Performance Computer Architecture

Instruction-Level Parallelism: Hardware Techniques

IIT KHARAGPUR

Instructor:
Prof. Rajat Subhra Chakraborty
Professor
Dept. of Computer Science and Engineering
Indian Institute of Technology Kharagpur
Kharagpur, West Bengal, India 721302
E-mail: rschakraborty@cse.iitkgp.ac.in

Recall: Approaches to Exploit ILP

- To increase processor throughput by overlapped instruction execution
  - Used in most modern processors (even low-end ones)
- Two main approaches:
  - Hardware Techniques (dynamic): parallelism found and exploited by hardware during execution
    - Used in most modern processors (e.g. x86-64, ARM, RISC-V)
  - Software Techniques (static): parallelism found and exploited by compiler during compilation
    - Used in most modern compilers (at least the basic techniques)
    - Sophisticated schemes have limited use (in domain-specific scientific applications only)
    - E.g. Intel Itanium processors (~1999)

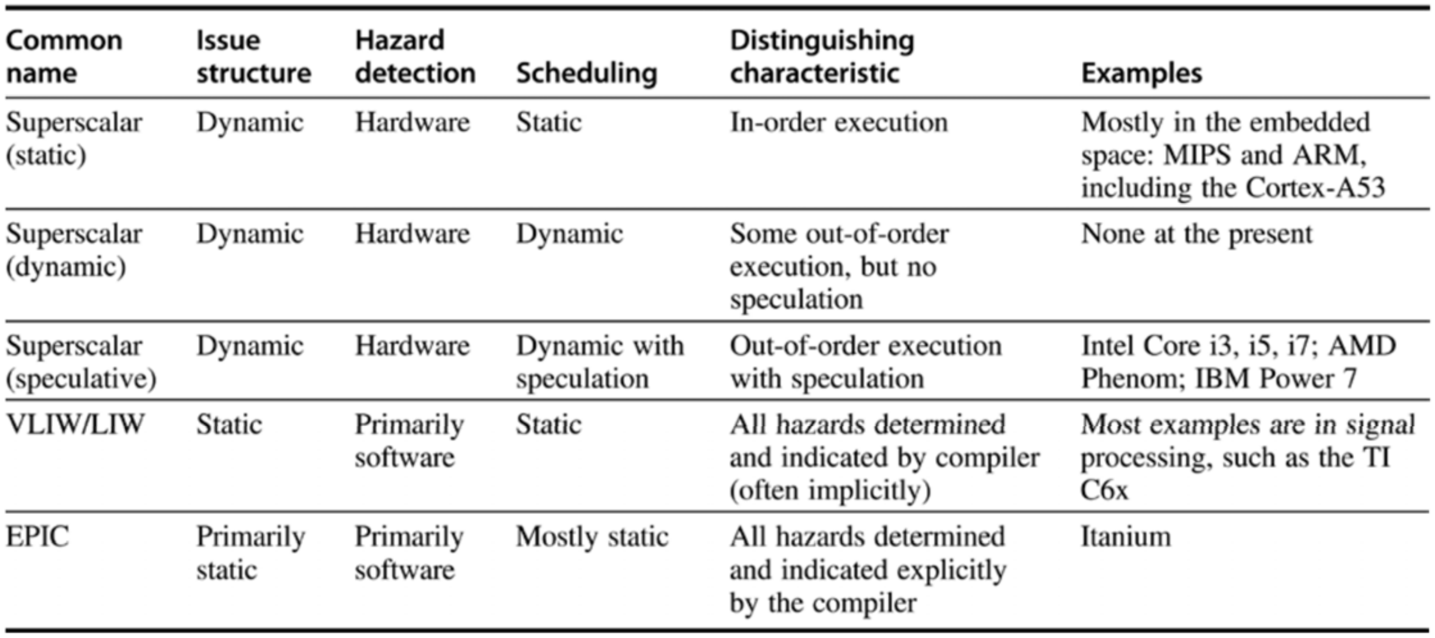
- In this lecture: we concentrate on basic and advanced hardware techniques to exploit ILP

Hardware Techniques to Exploit ILP

- **Pipelining: all modern processors**
- **Other techniques: (*superscalar*: multiple execution units)**

Static issue => fixed number of instructions issued per clock cycle
Dynamic issue => varying number of instructions issued per clock cycle

Static scheduling=> in-order execution
Dynamic scheduling => out-of-order execution



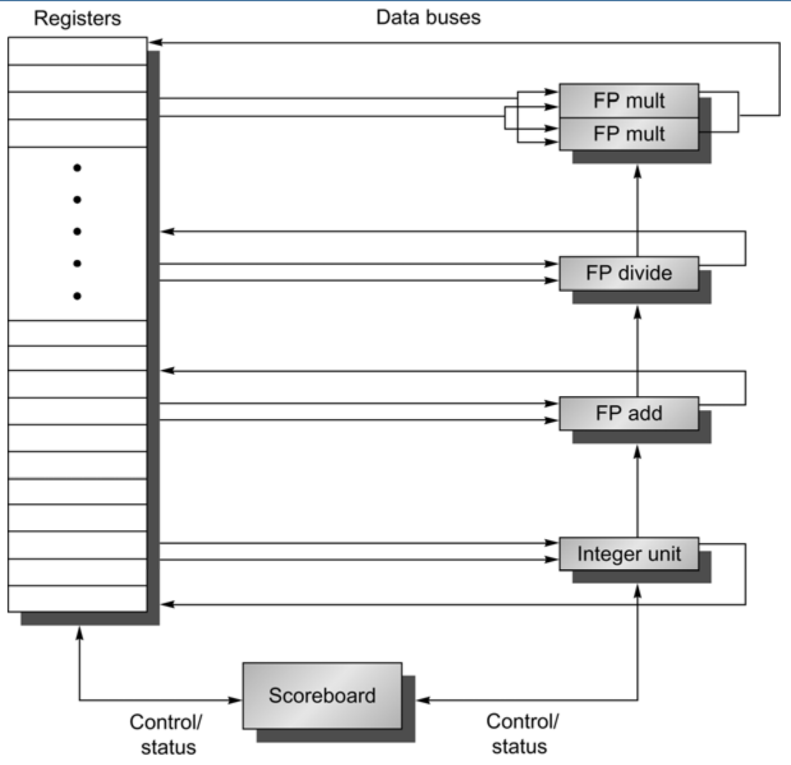
Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	Dynamic	Hardware	Static	In-order execution	Mostly in the embedded space: MIPS and ARM, including the Cortex-A53
Superscalar (dynamic)	Dynamic	Hardware	Dynamic	Some out-of-order execution, but no speculation	None at the present
Superscalar (speculative)	Dynamic	Hardware	Dynamic with speculation	Out-of-order execution with speculation	Intel Core i3, i5, i7; AMD Phenom; IBM Power 7
VLIW/LIW	Static	Primarily software	Static	All hazards determined and indicated by compiler (often implicitly)	Most examples are in signal processing, such as the TI C6x
EPIC	Primarily static	Primarily software	Mostly static	All hazards determined and indicated explicitly by the compiler	Itanium

H&P CA:A QA (6th. Ed.)

Dynamic Scheduling: *Scoreboard*

- Dynamic Scheduling: instructions can start and complete execution out-of-order
  - Instructions rearranged by hardware during run-time to avoid stalls
- Particularly suitable for relatively simple RISC instruction sets
  - Allows code to be scheduled to be executed efficiently by pipeline
  - Modern CISC instruction sets are RISC underneath!
    - A hardware layer translates CISC instructions to collection of RISC operations!
- Advantages of Dynamic Scheduling:
  - Allows the same compiled binary to run efficiently on different microprocessors with same instruction sets but different pipeline microarchitecture
    - Independent of compilation optimizations
  - Allows better handling of certain dependences which might be unknown at compile time
    - Due to memory reference, data-dependent branch, dynamic linking, etc.
  - Allows instructions to be executed when previous instructions are stalled
    - Because of: hazards, cache misses, etc.

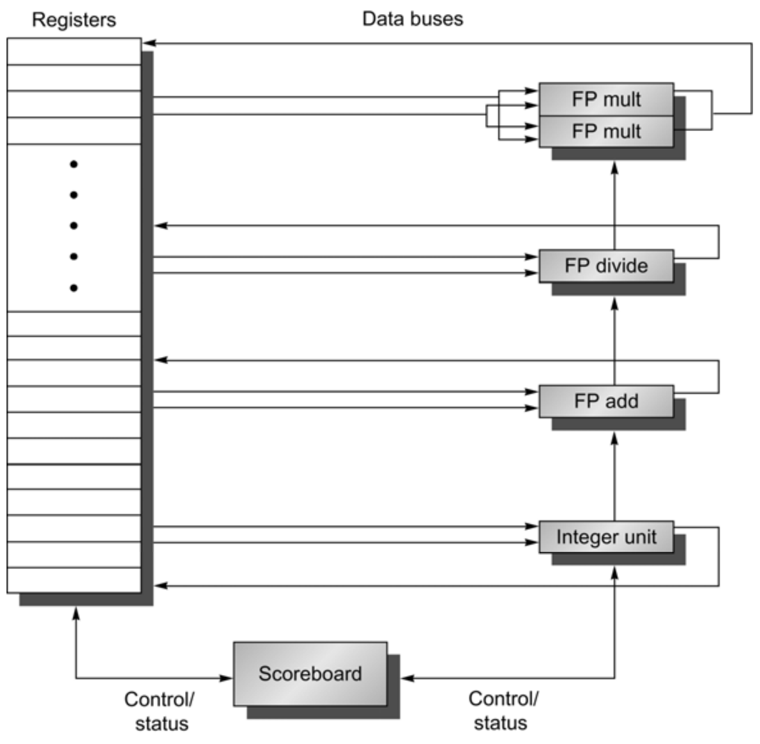
Example RISC-V Datapath with a Scoreboard



- ***Scoreboard* is a sophisticated technique for dynamic scheduling**
  - **Extremely advanced for its time**
  - **Great influence on later high-performance processor designs**
- **Features:**
  - **Used in the Control Data Corporation 6600 (CDC 6600) supercomputer (~1964)**
  - **CDC 6600 was the leading supercomputer till ~1969**
- **Note: CDC 6600 was a load-store architecture (similar to RISC-V)**

The basic structure of a (hypothetical) RISC-V processor with a scoreboard. The scoreboard's function is to control instruction execution (vertical control lines). All of the data flow between the register file and the functional units over the buses (the horizontal lines, called *trunks* in the CDC 6600). There are two FP multipliers, an FP divider, an FP adder, and an integer unit. One set of buses (two inputs and one output) serves a group of functional units.

RISC-V Datapath with a Scoreboard



Note: Functional units cannot read from or write to register file, unless allowed by scoreboard!

- **Systematically *avoids*: structural, WAW, RAW and WAR hazards**
- **Basic features:**
  - **Goal: to execute 1 instruction per clock cycle (provided there is no structural hazard)**
  - **Goal: to execute an instruction as early as possible**
  - **Instructions can possibly be stalled by hardware during run-time to avoid hazards**
  - **Instructions can possibly start and complete execution out-of-order**
  - **Instructions can possibly also update their results in register file out-of-order (!), if there is no WAR hazard**
  - **Instructions can possibly be rearranged by hardware during run-time to avoid stalls**

H&P CA:A QA (6th. Ed.)

Scoreboard: Steps of Operation

- **Slightly different from the traditional five steps of a classic RISC pipeline**
- **1. IF: gets feedback from ID stage, whether fetching should continue or stall**
- **2. Issue:**
  - **Issue stage operates in-order**
  - **Issue stage decodes instruction**
  - **If the necessary functional unit is free *and* there is no active instruction with same destination register, scoreboard issues the instruction to the functional unit, and notes the details internally**
  - **If *any* structural or WAW hazard exists, instruction issue to EX stalls**
    - **Instruction fetch might also eventually stall if queue between IF and Issue fills-up**
    - **Scoreboard resolves structural hazard in Issue stage**
    - **Scoreboard resolves WAW hazard in Issue stage**
    - **Issue resumes when structural and WAW hazards are cleared**

Scoreboard: Steps of Operation (contd.)

- 3. Read Operands:
  - Scoreboard monitors availability of source operands, i.e. no currently active instruction is going to write to it
  - When source operand is available, scoreboard allows functional unit to read the operands from the register file and begin execution
  - Instructions can possibly start execution out-of-order
  - Scoreboard resolves RAW hazards in this step
  - Note: Scoreboard-based system does not employ forwarding!

- 4. Execution:
  - Relevant execution unit starts execution
  - After one cycle (integer unit) or multiple cycles (FP unit) completes operation
  - Notifies Scoreboard when done
  - Similar to RISC EX stage

Scoreboard: Steps of Operation (contd.)

- **5. Write Result:**
  - **Scoreboard checks for WAR hazard for the recently completing instruction**
  - **If there is no WAR hazard, informs execution unit to update value in register file**
  - **If there is a WAR hazard, completing instruction is stalled**
    - **Once WAR hazard clears, completing instruction can update register file**
  - **Scoreboard resolves WAR hazards in this step**
  - **Note: Write Result and Read Operands cannot overlap! (as in MIPS/RISC-V)!**
  - **Example:**

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

fadd.d will *Issue* before fsub.d (since issue is in order)
but
fadd.d is dependent on fdiv.d
=> fadd.d will need to wait for many cycles in *Read Operands* stage!
Hence, fsub.d will bypass fadd.d in *Read Operands* stage, and complete execution before fadd.d!
Scoreboard will stall fsub.d *after execution* till *Read Operands* stage of fadd.d is complete!
After this, fsub.d will update f8!

Beyond Scoreboards: *Tomasulo's Algorithm*

- Scoreboards are not the best solution for modern processors
  - Dozens of execution units
  - Each execution unit deeply pipelined (e.g. > 10 pipeline stages in each unit)
  - Over 100 instructions being operated on ("in-flight") at any given instance
  - Scoreboard analysis limited to a basic block =>a small "window" for optimization

- *Tomasulo's Algorithm* [Tomasulo, IBM, 1967] is a more suitable and scalable solution
  - Even more influential than scoreboarding!
  - Much more complicated too (requires much more hardware resources)!
  - Similar schemes are used in all modern processors
  - Was first implemented on the IBM System/360 Model 91 FPU Units
  - IBM 360/91 was a direct competitor to the CDC 6600!

Scoreboarding vs. Tomasulo's Algorithm

- **Similarities:**
  - **Instructions are issued in order**
  - **Issue can stall completely if structural hazard exists for the given operation type**
  - **However, reading operands may be out-of-order**
  - **Instructions can complete writing to register file out of order**
- **Dissimilarities:**
  - **Scoreboaring: instruction issue stalls if there is possibility of WAW hazard**
  - **Scoreboarding: updating of register files is delayed if there is possibility of WAR hazard**
  - **Scoreboarding: does not support *speculation* (a technique to reduce the negative impact of control dependences)**
  - **Tomasulo's Algorithm: performs run-time (dynamic) *Register Renaming* during "Read Operands" phase to remove possibility of WAR and WAW hazards**
  - **Tomasulo's Algorithm: supports speculation**
  - **Tomasulo's Algorithm: complicated hardware (including additional buffers)**

Tomasulo's Algorithm: Goals

- **Seamlessly overcome the four types of hazards**

- **Overcome the limitations of compiler-assisted scheduling for IBM 360/91 FPU**
  - **The IBM 360/91 had only 4 FP registers!**

- **Overcome other architectural limitations of IBM 360/91:**
  - **Long memory access latency**
  - **Long floating-point operation latency**

- **Support *Speculative Execution* ("Speculation")**
  - **Guess what the branch outcome would be, and start executing**
  - **Even overlap consecutive loop iterations by predicting "Branch Taken"**
  - **Take corrective measures if prediction is found to be incorrect**

Recall: Name Dependences (WAR/WAW)

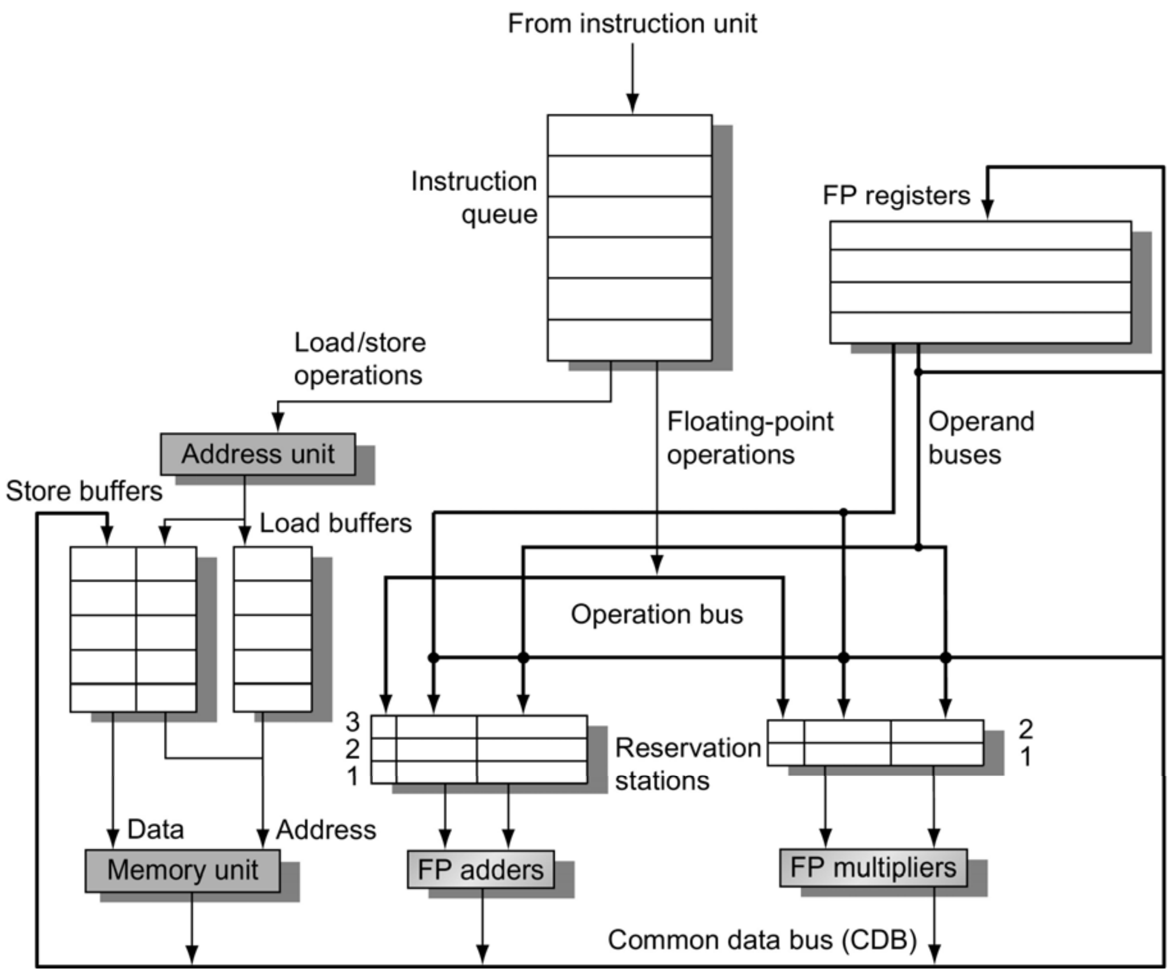
```
fdiv.d  f0,f2,f4
fadd.d  f6,f0,f8
fsd     f6,0(x1)
fsub.d  f8,f10,f14
fmul.d  f6,f10,f8
```

Potential WAW: fadd.d and fmul.d (on f6)
Potential WAR: fsub.d and fadd.d (on f8)
fsd and fmul.d (on f6)
Several potential RAW hazards!
<u>Solution</u>: Register renaming!
Assume: temporary registers S and T

Modified code (without any name dependence):
<u>Note:</u> True dependences are not removed!
<u>Advantage:</u> simple situations like these can be handled by a compiler
<u>Challenge:</u> finding S and T if number of available registers is low (e.g. in IBM 360/91 FPU)!
<u>Challenge:</u> all future use of f8 must be replaced by T!

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

Overall Architecture of Tomasulo's Algorithm for a FPU



H&P CA:A QA (6th. Ed.)

Important HW Units of Tomasulo's Algorithm for a FPU

1. **Instruction Queue**

   Instructions are sent from the instruction unit into the instruction queue from which they are issued in first-in, first-out (FIFO) order

2. **Reservation stations (RS)**

   Classic Tomasulo: One RS with one entry per Functional Unit (FU)
   Modern CPUs: Reservation station with multiple "rows" (entries) per FU
   More advanced: centralized single RS with very large number of entries, shared by multiple FUs
   (e.g. Intel Core i7: 97-entry centralized RS, shared by 6 functional units)
   Each reservation station entry: buffers the opcode and the actual operands for *one instruction*, as well as information used for detecting and resolving hazards
   Note: distributed functionality (different from Scoreboard!)

3. **Address Unit**

   Performs effective address computation for Load and Store operations

4. **Memory Unit**

   Interacts with main memory for load/store operations

Important HW Units of Tomasulo’s Algorithm for a FPU (contd.)

5. **Load Buffers**
   (a) hold the components of the effective address until it is computed (using “Address Unit”); (b) track outstanding loads that are waiting on the memory, and, (c) hold the results of completed loads that are waiting for the CDB.

6. **Store Buffers**
   (a) hold the components of the effective address until it is computed (using “Address Unit”), (b) hold the destination memory addresses of outstanding stores that are waiting for the data value to store, and (c) hold the address and value to store until the memory unit is available.

7. **Common Data Bus (CDB)**
   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.

Steps of Tomasulo's Algorithm

Three steps (*each* can take *arbitrary* clock cycles)*:*

1. ***Issue* (aka *Dispatch*)**

   (a) Get the next instruction from the head of the instruction queue, which is maintained in FIFO order to ensure the maintenance of correct data flow.

   (b) If there is a matching RS that is empty, issue the instruction to the station with the operand values, if they are currently in the registers.

   (c) If there is not an empty reservation station, then there is a structural hazard, and the instruction issue stalls until a station or buffer is freed.

   (d) If the operands are not in the registers, keep track of the functional units that will produce the operands.

   (e) **Perform register renaming if required, eliminating WAR and WAW hazards.**

Steps of Tomasulo's Algorithm (contd.): Non-LD/SD Instruction

2. Execute

(a) If one or more of the operands is not yet available, monitor the CDB while waiting for it to be computed.

(b) When an operand becomes available, it is placed into any reservation station awaiting it.

(c) When all the operands are available, execution starts at the corresponding functional unit.

Note: By delaying instruction execution until the operands are available, RAW hazards are avoided.

Note:

1. **If more than one instructions (in the Reservation Stations of a particular functional unit) becomes ready in the same clock cycle, the execution unit (except load/store) can choose arbitrarily among them.**
2. **(Very important!) If successive writes to a register overlap in execution, *only the last one* is actually used to update the register!**

Elimination of WAR Hazard in Tomasulo's Algorithm

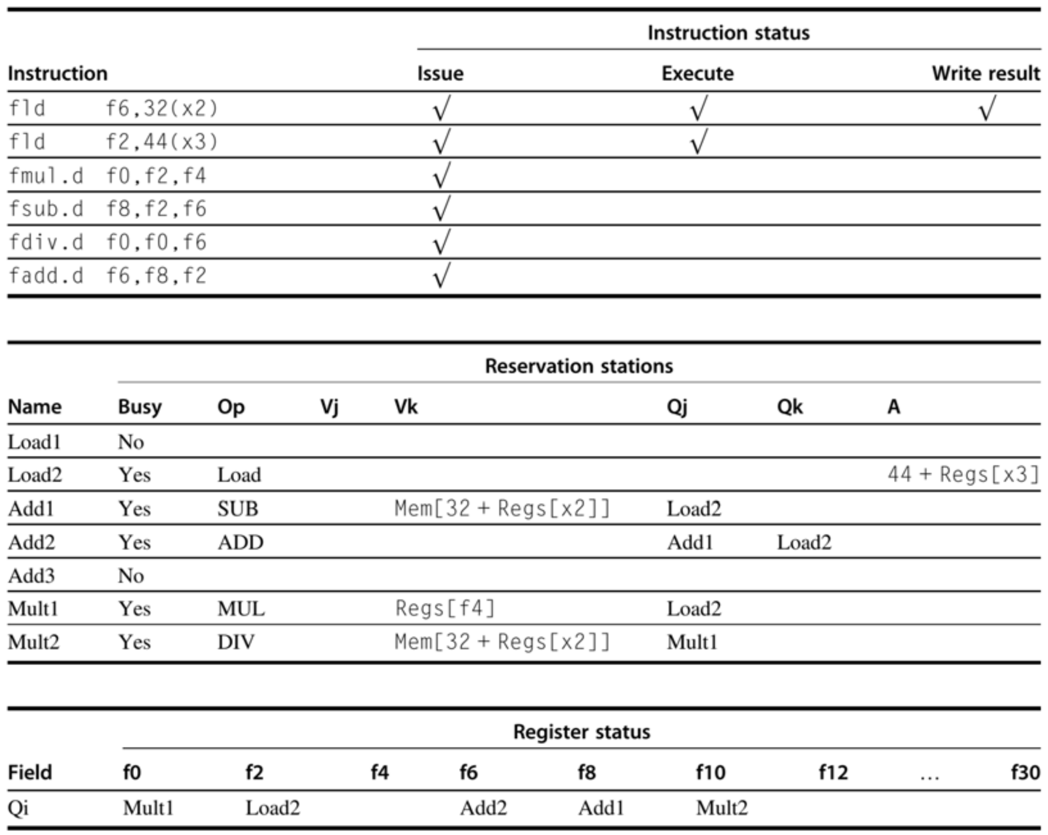
```
fld     f6,32(x2)
fld     f2,44(x3)
fmul.d  f0,f2,f4
fsub.d  f8,f2,f6
fdiv.d  f0,f0,f6
fadd.d  f6,f8,f2
```

- fadd.d has WAR on fdiv.d (reason: f6)
- Can fadd.d issue and complete WB before fdiv.d?: Yes!!
- Note: f6 is also generated by fld before fdiv.d!
- Situation: fdiv.d is in its RS

  Consider three cases:
- Case-1: fld f6, 32(x2) has completed WB even before fdiv.d starts
  - RS for fdiv.d receives value from Register File during Issue itself
- Case-2: fld f6, 32(x2) completes execution, but fadd.d has not yet completed execution
  - RS for fdiv.d receives a copy of the value fetched from memory through CDB
  - fdiv.d is not affected by changes made by fadd.d on f6 later!
- Case-3: fld f6, 32(x2) has not completed execution, fadd.d completes execution
  - RS for fdiv.d points to fld for operand
  - RS for fdiv.d receives a copy of this result from through CDB *eventually* when fld completes
  - fadd.d can complete WB out-of-order, before fdiv.d has completed!
  - fld is not allowed to update Register File, i.e. cannot perform WB on f6!

Elimination of WAR Hazard in Tomasulo's Algorithm (contd.)

Case-1 from last slide shown here (1st fld has completed WB)!



Instruction		Instruction status		
		Issue	Execute	Write result
`fld`	`f6,32(x2)`	√	√	√
`fld`	`f2,44(x3)`	√	√	
`fmul.d`	`f0,f2,f4`	√		
`fsub.d`	`f8,f2,f6`	√		
`fdiv.d`	`f0,f0,f6`	√		
`fadd.d`	`f6,f8,f2`	√		

	Reservation stations						
Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	No						
Load2	Yes	Load					`44 + Regs[x3]`
Add1	Yes	SUB		`Mem[32 + Regs[x2]]`	Load2		
Add2	Yes	ADD			Add1	Load2	
Add3	No						
Mult1	Yes	MUL		`Regs[f4]`	Load2		
Mult2	Yes	DIV		`Mem[32 + Regs[x2]]`	Mult1		

	Register status							
Field	f0	f2	f4	f6	f8	f10	f12	…	f30
Qi	Mult1	Load2		Add2	Add1	Mult2			

Figure 3.11 Reservation stations and register tags shown when all of the instructions have issued but only the first load instruction has completed and written its result to the CDB. The second load has completed effective address calculation but is waiting on the memory unit. We use the array Regs[ ] to refer to the register file and the array Mem[ ] to refer to the memory. Remember that an operand is specified by either a Q field or a V field at any time. Notice that the `fadd.d` instruction, which has a WAR hazard at the WB stage, has issued and could complete before the `fdiv.d` initiates.

Elimination of WAW Hazard in Tomasulo's Algorithm

- fadd.d has WAW on fld (reason: f6)
- Can fadd.d issue and complete WB before fld.d?: Yes!!
  - fld f6, 32(x2) gets the value from memory, passes it to dependent instruction through CDB, but does not update Register File for f6!
- Question: fdiv.d also has WAW on fmul.d….is it something of concern?

```
fld     f6,32(x2)
fld     f2,44(x3)
fmul.d  f0,f2,f4
fsub.d  f8,f2,f6
fdiv.d  f0,f0,f6
fadd.d  f6, f5 ,f2
```

Steps of Tomasulo’s Algorithm (contd.): LD/SD Instruction

LD/SD: two-step process:

Step-1:

(a) Calculate effective address (using “Address Unit”), when base register content is available

(b) **Effective address calculation is performed in program order**

(c) Place effective address in RS

Step-2 (see next slide also):

(a) LD: execute as soon as memory unit is available.

(b) SD: wait for value to stored to be available, then send to memory as soon as available

Note:

(a) By delaying effective address calculation, RAW hazards are avoided.

(b) For SD, by delaying write to memory, RAW hazards are avoided.

(c) By calculating effective address in program order, hazards through memory are avoided

e.g.
```
sd f3, 100(x1)
ld  f2, 100(x1)
```
To avoid hazard, ld should not read from memory before sd completes writing to memory!

Steps of Tomasulo’s Algorithm (contd.)

3. Write Result

(a) When the result is available, write it on the CDB and from there into the registers and into any reservation stations (including store buffers) waiting for this result.

(b) For SD: Stores are buffered in the store buffer until both the value to be stored and the store address are available; then the result is written as soon as the memory unit is free.

Note: the result after generation by the corresponding functional unit of retrieval from memory is written in corresponding RS or Load Buffer. From there, in the next clock cycle it is written to RS or Store Buffer waiting for it through the CDB. And then in the next clock cycle, it can be used in functional unit.

Hence, there is 1 clock cycle latency between production and usage of a value.

Advantages of Tomasulo’s Algorithm Over Scoreboarding:

1. Distributed hazard detection and execution control.
2. Automatic hardware-based elimination of WAR/WAW hazards.
3. Not limited to basic blocks (see next).
4. Automatic forwarding/bypassing-type effect because of CDB (although there is one additional cycle latency compared to simple pipeline).
5. Allow loop unrolling in HW.

What About Exceptions?

Recall: we want *Precise Exceptions*

Soln.-1 (Defensive): An instruction is not allowed to start execution *until a branch instruction before it has completed*.

Hence, if branch prediction is used, it must be known that the result of branch prediction is correct, before a later instruction is allowed to execute.

Soln.-2 (Advanced): The processor notes the exception, but does not raise it.

At the end of execution, the instruction which caused the exception is not allowed to update the Register File!

Tomasulo’s Algorithm: “Virtual Registers”

- Data structures are attached to the RSs, the Load/Store Buffers, the Register File
- **The RSs and the Load buffers act as an extended set of “virtual registers”**
    - These “virtual registers” are not visible to the application programmer!
    - Represented by numerical “tags” in the data structures
    - E.g. 5 (RSs) + 5 (Load Buffer) = 10 additional buffers => represented by 4-bit tag
    - These registers can be used for register renaming, by being designated as result registers
- In modern high-end processors, the number of “virtual registers” can be in hundreds!
    - **Much more RSs than architectural registers => renaming is easy!**

- In the data structures, there is no mention of the “architectural registers”!
    - **Register names are discarded when an instruction is issued to a RS!**
    - Different from Scoreboard!
    - **Hence, renaming is automatic and implicit**

- A source operand already available in register file: referred to by 0
- A source operand to be produced by another instruction: referred to by the reservation station number attached to the instruction that will produce it!

Can We Reorder Loads and Stores?

- If Loads and Stores target different addresses, they can be executed out-of-order
- **However, if Load/Store and Store/Store share the same address, reordering them can be problematic!**

Example-1: SD before LD

```
fsd f1, 0(x1)
fld  f0, 0(x1)
```

$\xrightarrow{\text{Reorder execution}}$

```
fld   f0, 0(x1)
fsd  f1, 0(x1)
```

New RAW hazard introduced (artificially!), since fld will read the old (non-updated) value from memory!

Example-2: LD before SD

```
fld   f0, 0(x1)
fsd  f1, 0(x1)
```

$\xrightarrow{\text{Reorder execution}}$

```
fsd   f1, 0(x1)
fld    f0, 0(x1)
```

New WAR hazard introduced (artificially!), since fld will read the updated (incorrect) value from memory!

Example-3: SD before SD

```
fsd   f0, 0(x1)
………….
fsd   f1, 0(x1)
```

$\xrightarrow{\text{Reorder execution}}$

```
fsd   f1, 0(x1)
……………….
fsd    f0, 0(x1)
```

New WAW hazard introduced (artificially!), since memory will have incorrect value!

Solution: Rules To be Followed

- **<u>For Loads:</u>**
  - **Wait till a waiting Store sharing same address before it in program completes**
  - **Load/Load can be freely reordered!**
- **<u>For Stores:</u>**
  - **Wait till a waiting Load/Store sharing same address before it in program completes**
- **<u>Simple solution:</u>**
  - **Perform effective address computation in program order**
    - **All Load/Store Buffer entries have resolved addresses**
  - **<u>For Load Buffer Entries</u>: Check for conflicts in Store Buffers**
  - **<u>For Store Buffer Entries</u>: Check for conflicts in *both* Load and other Store Buffers**

How Effective Are Dynamically Scheduled Processors?

Basic Idea:

- **Maintain a large set of additional buffers**
- **Accompanied by complex control circuitry**
- **Issue instruction (if no structural hazard), perform register renaming *implicitly***
- **Sometimes, wait till hazards are resolved**

Advantages:

- **They can give very high throughput, if combined with accurate branch prediction!**
- **Performs well under unpredictable cache delay**
- **Extends easily to *wide issue* processors (multiple instructions issued per clock cycle)**
- **Effective for difficult to schedule codes**
- **Compiler-independent, hardware-based optimizations**

- **Disadvantage:**
  - **Complexity and high hardware cost**

- **Tomasulo's Scheme was not used for a long time because of hardware costs**
- **However, similar schemes are widely used since 1990s**

Going Further: Hardware-based Speculation (a.k.a. *Speculative Execution*)

Motivation:

- **Accurate branch prediction insufficient to reach higher IPC, if a processor issues multiple instructions per clock cycle ("wide-issue processor")**
- **Perform out-of-order execution of multiple basic blocks *across branches*!**

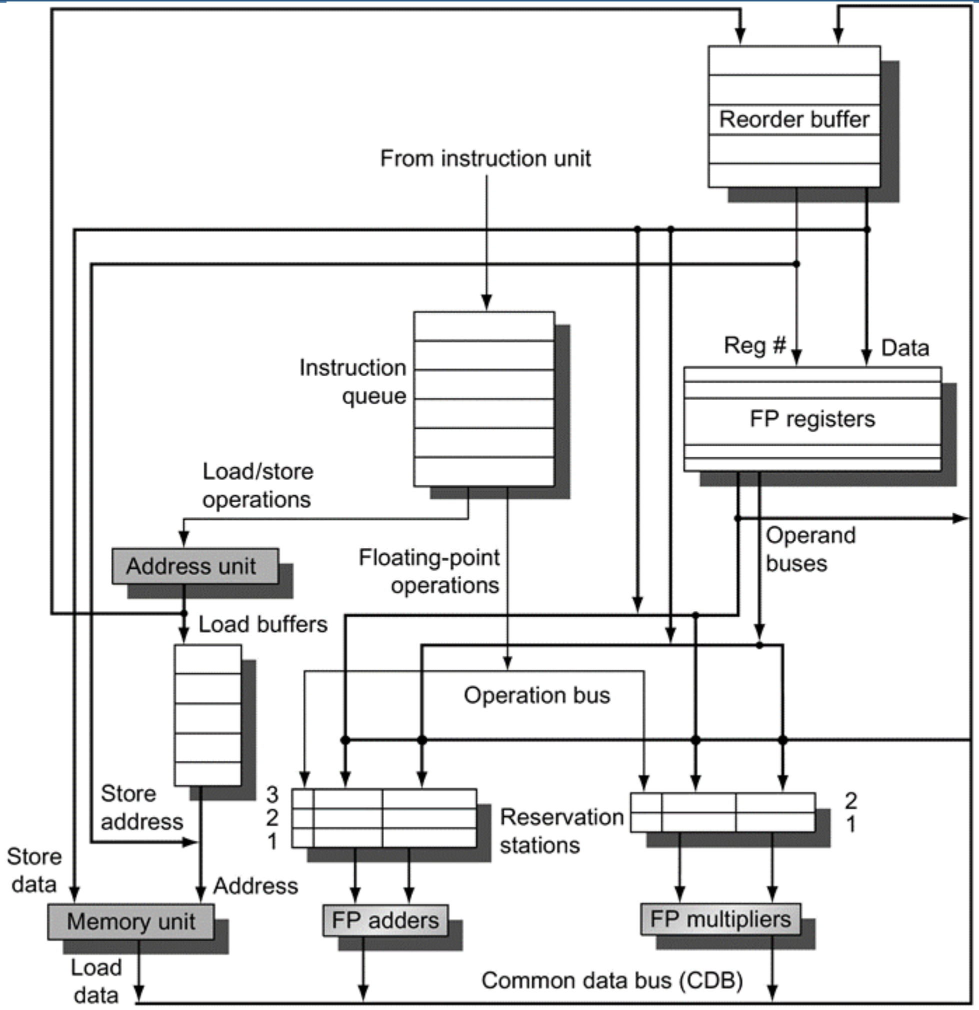
Dynamic Scheduling *without Speculation*

- **Fetch and issue instructions based on branch prediction, but do not execute them**
- **Can only *partially overlap* basic blocks (e.g. up to 2 instructions from next loop iteration)**
  - **Before executing, must know that the branch prediction was correct!**

Dynamic Scheduling *with Hardware-based Speculation*

- **Fetch, issue *and execute* instructions based on branch prediction!**
  - **Start execution from predicted portion *even before branch has been resolved*!**
  - **As if it knows that the *predicted branch will be correct*!**
  - **If finally it is found that predicted branch is incorrect, *undo the effects of the incorrectly executed instructions*!**

Tomasulo’s Algorithm with Speculation: Microarchitecture



H&P CA:A QA (6th. Ed.)

Tomasulo's Algorithm with Speculation: Key Ideas

- A new step introduced: ***Instruction Commit***
  - Register file/memory is updated
  - Happens ***only after the branch has been resolved to be correct*!**
- After execution finishes, result waits at a buffer called the **Reorder Buffer (ROB)**
- The ROB:
  - Guarantees **In-order commit** to update register file/memory
  - Subsumes Store Buffer (i.e. no separate Store Buffer)
  - Passes operand values to a RS, if such operand values are not yet in register file, but a RS needs them

- In **Tomasulo's scheme *without speculation***:
  - Computed values are found in register file from one clock cycle after the execution has completed
- In **Tomasulo's scheme *with speculation***:
  - Computed values are found in ROB from one clock cycle after the execution has completed
  - These values are found in register file ***only after instruction commit has completed*!**

Tomasulo’s Algorithm with Speculation: More about ROBs

- The **ROB performs register renaming** (in place of Reservation Station)
  - Other roles of the Reservation Stations (RSs) remain unchanged
- After execution finishes, result from Functional Unit is sent via CDB to ROBs and other RSs waiting for it

- RSs read operand values from
  - Register file (if operand is available)
  - From another RS (through CDB based “forwarding” of operand value)
  - From appropriate ROB (If operand is not yet in Register File, but waiting in a ROB)

- **RS refers to destination by the ROB** which is handling the relevant instruction

- Since Instruction Commit is in-order, computed result may wait in ROB for many clock cycles before being written to register file/memory!

- **In-order instruction commit ensures *Precise Interrupt*!**

Tomasulo’s Algorithm with Speculation: Steps

- **1. Instruction Issue:**
  - **In-order issue, if there is an empty matching RS (as before), and empty ROB (to hold the result after EX)**
  - **Send the operand to RS if available in register, or in some other ROB (where it is waiting before commit). RS entry notes destination ROB after EX**
  - **Mark RS and destination ROB *busy***
  - **No matching RS and/or no free ROB available => structural hazard => issue stalls!**

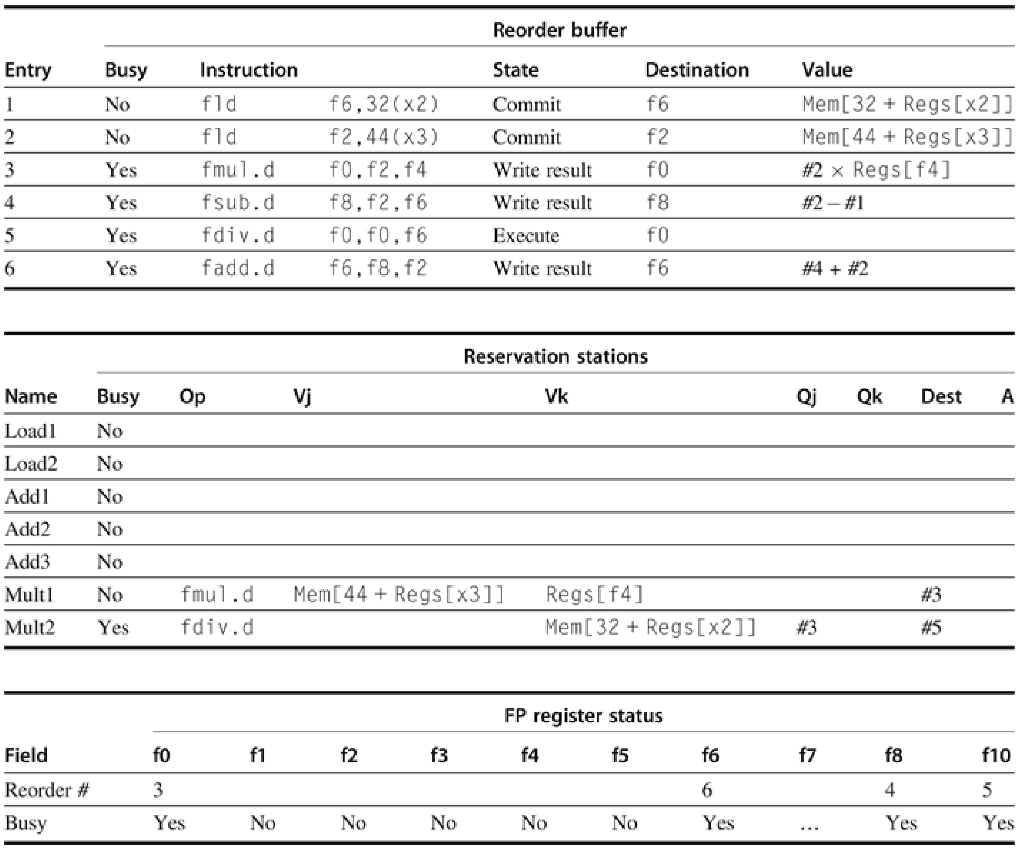
- **2. Execute:**
  - **Wait till all operands are available (keep monitoring CDB)**
  - **Start EX at functional unit when all operands are available**
  - **Note: Stores can start to calculate the effective address, data value can be obtained later!**

Tomasulo’s Algorithm with Speculation: Steps (contd.)

- **3. Write Result:**
  - **Write result (accompanied by destination ROB tag) on CDB when computed**
  - **Mark the RS *available***
  - **Note: if the Store data value in not yet ready, ROB entry is still not complete.**
    - **On every instruction completion, scan ROB for dependent stores, and update ROB**

- **4. Commit (a.k.a. “graduation”/completion”):**
  - **Commit happens in program order => ROB entries may need to wait for many clock cycles!**
  - **Normal commit: update register file/memory once instruction reaches head of ROB**
    - **Mark ROB *available***
  - **If a branch with correct prediction reaches head of ROB:**
    - **Branch is done (nothing else to do)**
  - **If a branch with incorrect prediction reaches head of ROB:**
    - **Flush ROB!**
    - **Since register file/memory is yet to be updated, processor “state” does not change!**
    - **Start execution at correct branch destination!**

Tomasulo's Algorithm with Speculation: Example



Reorder buffer						
Entry	Busy	Instruction		State	Destination	Value
1	No	fld	f6,32(x2)	Commit	f6	Mem[32 + Regs[x2]]
2	No	fld	f2,44(x3)	Commit	f2	Mem[44 + Regs[x3]]
3	Yes	fmul.d	f0,f2,f4	Write result	f0	#2 × Regs[f4]
4	Yes	fsub.d	f8,f2,f6	Write result	f8	#2 – #1
5	Yes	fdiv.d	f0,f0,f6	Execute	f0	
6	Yes	fadd.d	f6,f8,f2	Write result	f6	#4 + #2

Reservation stations								
Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
Load1	No							
Load2	No							
Add1	No							
Add2	No							
Add3	No							
Mult1	No	fmul.d	Mem[44 + Regs[x3]]	Regs[f4]			#3	
Mult2	Yes	fdiv.d		Mem[32 + Regs[x2]]	#3		#5	

FP register status										
Field	f0	f1	f2	f3	f4	f5	f6	f7	f8	f10
Reorder #	3						6		4	5
Busy	Yes	No	No	No	No	No	Yes	...	Yes	Yes

fadd.d/fsub.d: 2 clock cycles
fmul.d: 6 clock cycles
fdiv.d: 12 clock cycles

Figure 3.16 At the time the `fmul.d` is ready to commit, only the two `fld` instructions have committed, although several others have completed execution. The `fmul.d` is at the head of the ROB, and the two `fld` instructions are there only to ease understanding. The `fsub.d` and `fadd.d` instructions will not commit until the `fmul.d` instruction commits, although the results of the instructions are available and can be used as sources for other instructions. The `fdiv.d` is in execution, but has not completed solely because of its longer latency than that of `fmul.d`. The Value column indicates the value being held; the format #X is used to refer to a value field of ROB entry X. **Reorder buffers 1 and 2 are actually completed (empty) but are shown for informational purposes**. We do not show the entries for the load queue, but these entries are kept in order.

H&P CA:A QA (6th. Ed.)

Tomasulo’s Algorithm with Speculation: Loop Example

Reorder buffer

Entry	Busy	Instruction		State	Destination	Value
1	No	`fld`	`f0,0(x1)`	Commit	`f0`	`Mem[0 + Regs[x1]]`
2	No	`fmul.d`	`f4,f0,f2`	Commit	`f4`	#1 × `Regs[f2]`
3	Yes	`fsd`	`f4,0(x1)`	Write result	`0 + Regs[x1]`	#2
4	Yes	`addi`	`x1,x1,−8`	Write result	`x1`	`Regs[x1] – 8`
5	Yes	`bne`	`x1,x2,Loop`	Write result		
6	Yes	`fld`	`f0,0(x1)`	Write result	`f0`	`Mem[#4]`
7	Yes	`fmul.d`	`f4,f0,f2`	Write result	`f4`	#6 × `Regs[f2]`
8	Yes	`fsd`	`f4,0(x1)`	Write result	0 + #4	#7
9	Yes	`addi`	`x1,x1,−8`	Write result	`x1`	#4 – 8
10	Yes	`bne`	`x1,x2,Loop`	Write result		

FP register status

Field	f0	f1	f2	f3	f4	F5	f6	F7	f8
Reorder #	6								
Busy	Yes	No	No	No	Yes	No	No	…	No

fadd.d/fsub.d: 2 clock cycles
fmul.d: 6 clock cycles
fdiv.d: 12 clock cycles

Figure 3.17 Only the `fld` and `fmul.d` instructions have committed, although all the others have completed execution. Thus no reservation stations are busy and none are shown. The remaining instructions will be committed as quickly as possible. **The first two reorder buffers are empty, but are shown for completeness**.

How are Hazards Avoided?

- Ordinary WAR/WAW Hazards: avoided by register renaming in ROB

- WAR/WAW hazards through memory:
  - Are avoided since memory updates happen in-order
  - When a SD is at the head of the ROB, no other LD/SD are pending!

- Ordinary RAW hazard: avoided by making instructions wait in RS, as long as required

- RAW hazards through memory:
  - Perform effective address calculation of all LDs later with respect to all earlier SDs
  - Do not allow LD to access memory if an active ROB containing a SD has the same Destination Address as the LD’s Source Address
  - Better Alternative: bypass value from SD to LD waiting for the value (even before SD has completed memory update)

Extension to Dynamically Scheduled Multiple-issue Superscalar

- Issue multiple instructions per clock cycle
    - A collection of instructions issued in a clock cycle: *issue bundle*
- Issuing multiple instructions per clock cycle is complicated!
    - The multiple instructions being issued may have mutual dependences!
    - The tables must be updated in parallel, for the multiple instructions being issued!
    - This complication exists even in single-issue processors!
    - *n* instructions in a bundle => $O(n^2)$ dependencies to be analyzed and handled within one clock cycle!
    - Current issue rate: 4-8 instructions issued per clock cycle
- Approaches:
- 1st Approach: issue one instruction at *each of rising and falling* clock edge!
    - Limitation: maximum 2 instructions can be issued per clock cycle!
- 2nd Approach: make the issue hardware complex
    - e.g.: issue from multiple heads of the issue unit, each issue unit is pipelined!
    - Note: Simply deeply pipelining the issue unit to make it issue process fast will not work, since the RS/ROB updates must also need to happen equally fast!

Dynamically Scheduled Multiple-issue Superscalar: Microarchitecture

The following components should be widened (not shown):

- **the CDB**
- **the issue logic**
- **the operand buses**

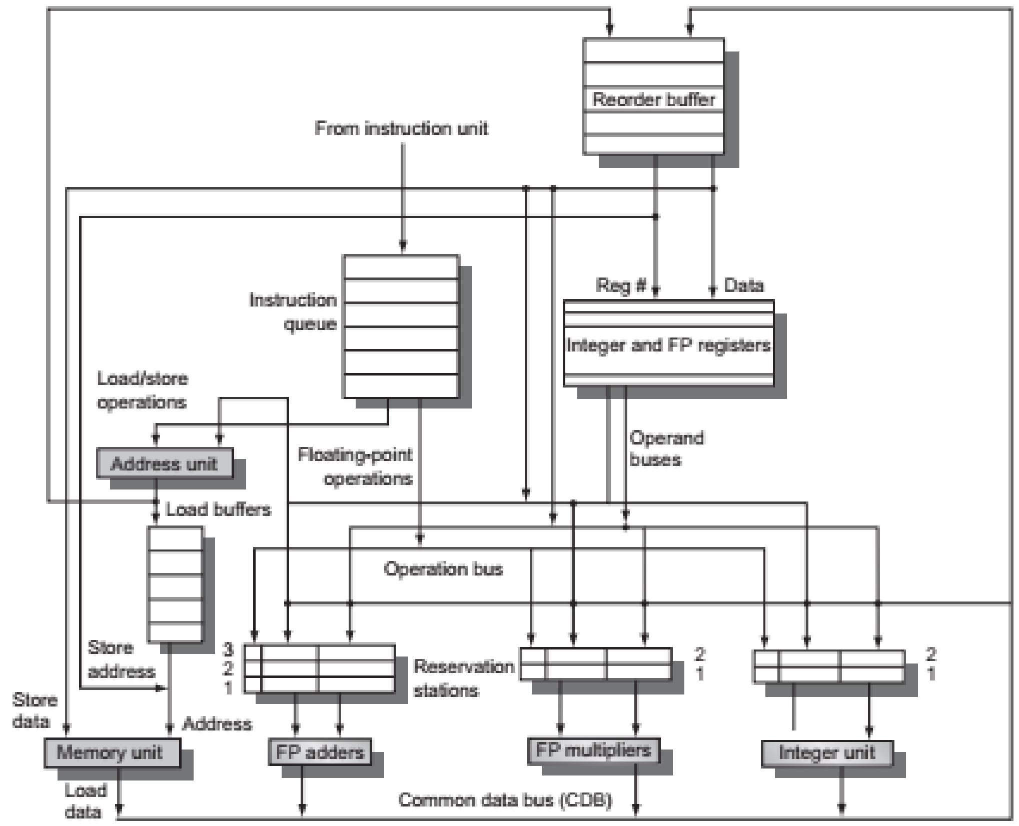


Figure 3.21 The basic organization of a multiple issue processor with speculation. In this case, the organization could allow a FP multiply, FP add, integer, and load/store to all issues simultaneously (assuming one issue per clock per functional unit). Note that several datapaths must be widened to support multiple issues: the CDB, the operand buses, and, critically, the instruction issue logic, which is not shown in this figure. The last is a difficult problem, as we discuss in the text.

H&P CA:A QA (6th. Ed.)

Dynamically Scheduled Multiple-issue Superscalar: Branch Target Buffer (BTB)

- **Predicted Branch Target Address must be known even before it is known that the instruction being decoded is a branch!**
- **Branch Target Buffer (BTB) is a special small fully associative cache**
- **BTB stores *known* branch instruction addresses which are *predicted taken* (T)**
  - **The corresponding Predicted Target Address is stored by its side**
- **No need to store predicted not taken (NT) branch addresses!**
  - **The target address is the sequential address**

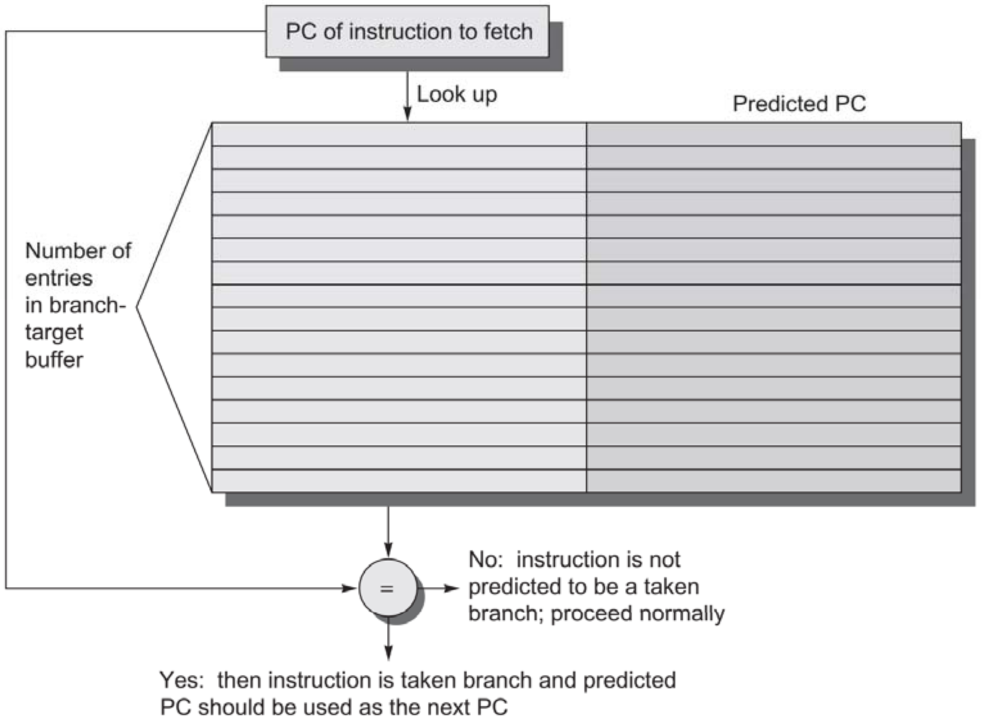

PC of instruction to fetch
Look up
Predicted PC
Number of entries in branch-target buffer
=
No: instruction is not predicted to be a taken branch; proceed normally
Yes: then instruction is taken branch and predicted PC should be used as the next PC

Figure 3.25 A branch-target buffer. The PC of the instruction being fetched is matched against a set of instruction addresses stored in the first column; these represent the addresses of known branches. If the PC matches one of these entries, then the instruction being fetched is a taken branch, and the second field, predicted PC, contains the prediction for the next PC after the branch. Fetching begins immediately at that address. The third field, which is optional, may be used for extra prediction state bits.

Branch Target Buffer (BTB) Usage Flowchart

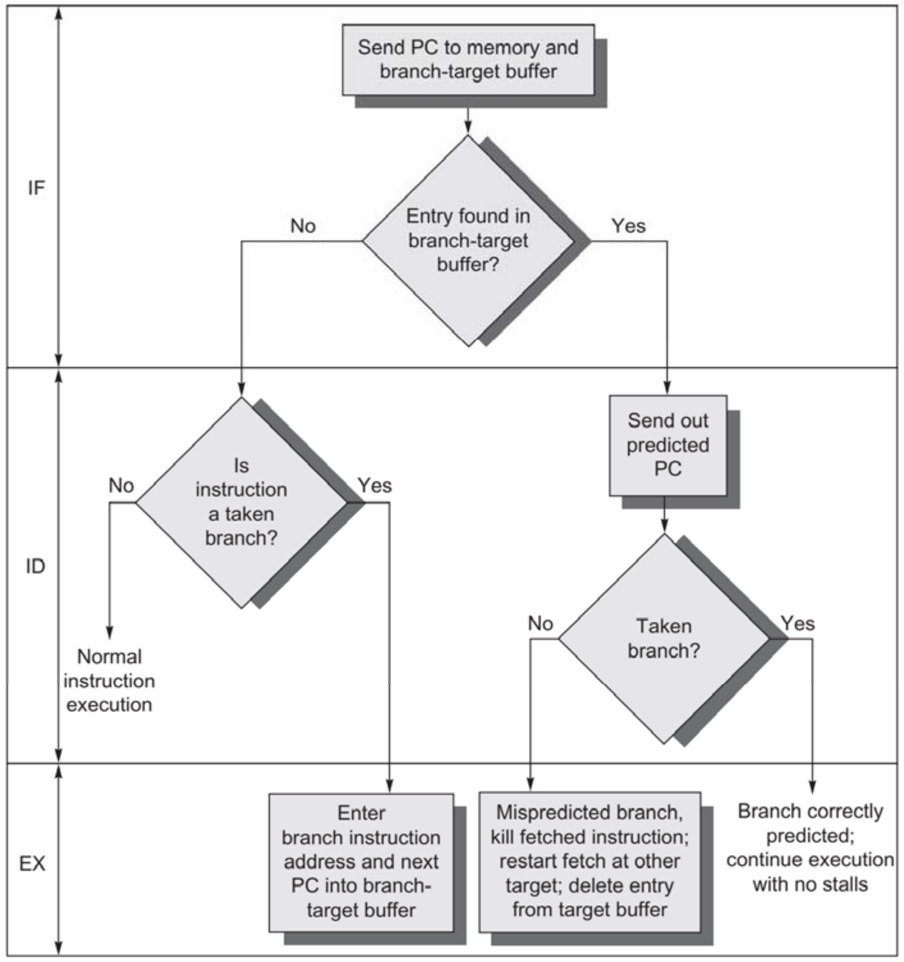

IF
ID
EX
Send PC to memory and branch-target buffer
Entry found in branch-target buffer?
No
Yes
Is instruction a taken branch?
No
Yes
Normal instruction execution
Send out predicted PC
Taken branch?
No
Yes
Enter branch instruction address and next PC into branch-target buffer
Mispredicted branch, kill fetched instruction; restart fetch at other target; delete entry from target buffer
Branch correctly predicted; continue execution with no stalls

Figure 3.26 The steps involved in handling an instruction with a branch-target buffer.

H&P CA:A QA (6th. Ed.)

Branch Target Buffer Application: *Branch Folding*

- **Expand BTB to also hold the predicted instruction along with the predicted address!**
- **Make BTB more sophisticated to be able to detect whether the branch is unconditional!**
  - **Not that difficult (very few unconditional branch instruction variants!)**
  - **A simple comparison of the instruction opcode will do!**
- **Consider an *unconditional branch***
  - **Observation: The target at the branch target will *always execute*!**
  - **Optimization step: as soon as an unconditional branch instruction is found in BTB after IF, pipeline reads the BTB to start ID on the branch target instruction, and abandons the current unconditional branch instruction (from ID onwards)!**
  - **This optimization is called *Branch Folding***
- **Results:**
  - **effort saved for the unconditional branch (no ID, EX, MEM, WB steps!)**
  - **effort saved for the branch target instruction (no IF required for it, read directly from BTB!)**

- **Might be possible to extend this optimization for some conditional branches also!**
  - **If it is *somehow known before ID* that the conditional branch will be Taken!**

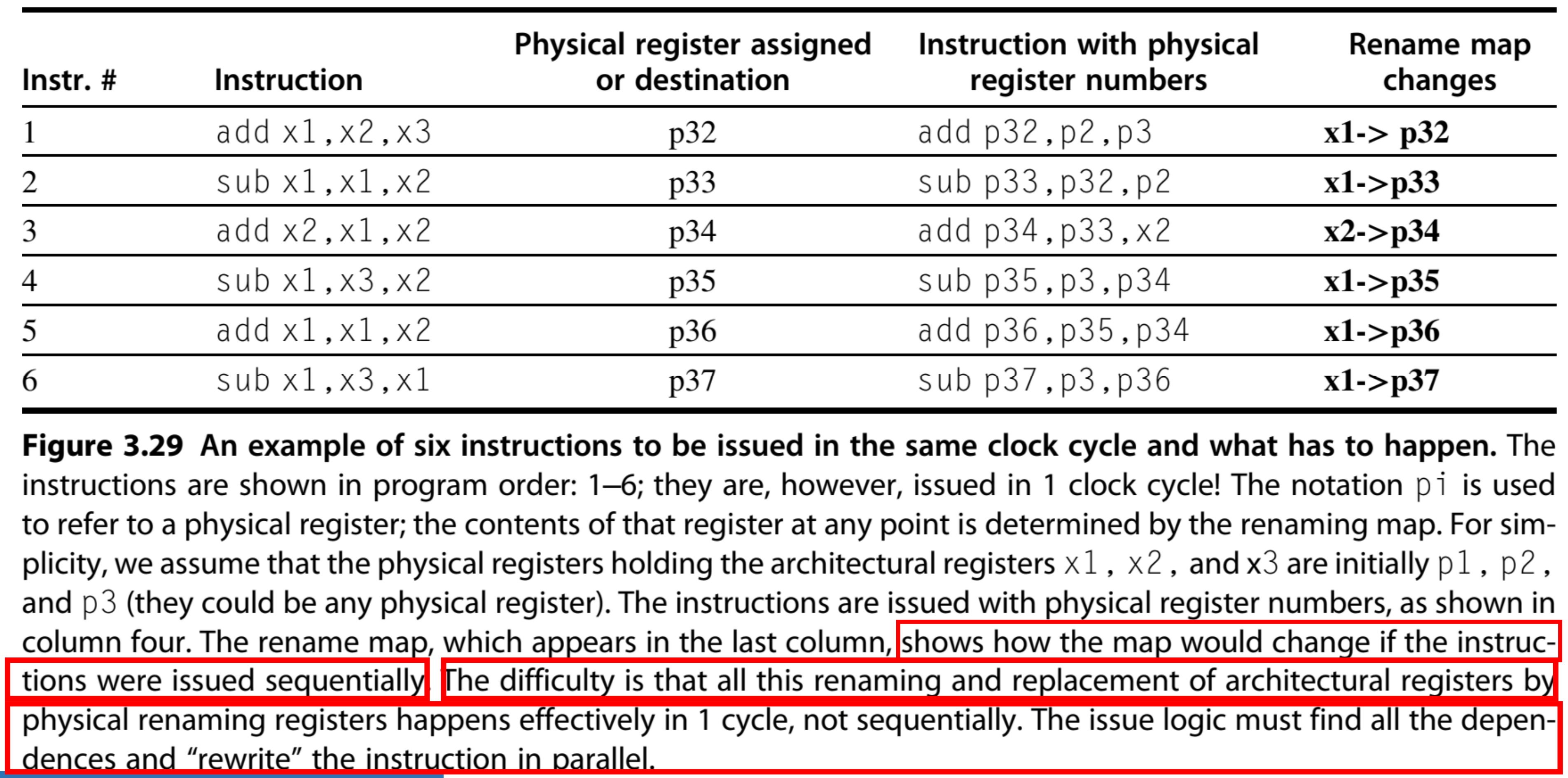
Dynamically Scheduled Multiple-issue Superscalar: Supporting/Alternative Techniques

- **Specialized Branch Predictors**
  - **Procedure Returns**
    - **Maintain a small stack of return addresses (push on call, pop on return)**
  - **Indirect Jumps (jumps whose destination varies from run-to-run):**
    - **e.g. switch…case statements, indirect procedure calls**
  - **Loop branch predictor**
- **Sophisticated IF Unit (with additional functionality)**
  - **Integrated Branch Prediction in IF unit**
  - **Instruction Prefetch**
    - **Fetch multiple (e.g. two) blocks while fetching blocks to instruction cache**
  - **Instruction Buffering to Enable Access to Instructions Distributed Across Cache Blocks**
- **Register Renaming with Additional (non-architectural) Registers**
  - **Before EX => Avoids WAR/WAW hazards**
- **"Renaming map": maintains renaming information**
  - **Alternative to ROBs, but modern processors use both approaches!**

Renaming Map (RM): Details

- Consider dynamically scheduled wide-issue speculative superscalar processor
    - Multiple instructions are speculatively issued every clock cycle, with dynamic register renaming
    - Multiple physical registers may be linked with an architectural register at any given point
    - RM records all these multiple mappings *provisionally*
    - RM needs to know that an instruction's execution is correct before final decision
    - Once an instruction commits (i.e. it has become *non-speculative*), the RM entry may be marked to be final!
- The register mappings keep on changing continuously in the RM!
    - Register state saving on context switch becomes very complicated!
    - "Old" RM entries need to be deleted/invalidated!
    - The physical registers holding "old" values must be freed!
- How does this happen for an instruction bundle that has been issued?
    - Extremely complicated to manage in a single clock cycle
    - Limits *issue width* to be usually between 4-8 instructions per clock cycle

Renaming Map (RM): Example Operation in an Instruction Bundle



Instr. #	Instruction	Physical register assigned or destination	Instruction with physical register numbers	Rename map changes
1	`add x1,x2,x3`	p32	`add p32,p2,p3`	**x1-> p32**
2	`sub x1,x1,x2`	p33	`sub p33,p32,p2`	**x1->p33**
3	`add x2,x1,x2`	p34	`add p34,p33,x2`	**x2->p34**
4	`sub x1,x3,x2`	p35	`sub p35,p3,p34`	**x1->p35**
5	`add x1,x1,x2`	p36	`add p36,p35,p34`	**x1->p36**
6	`sub x1,x3,x1`	p37	`sub p37,p3,p36`	**x1->p37**

Figure 3.29 An example of six instructions to be issued in the same clock cycle and what has to happen. The instructions are shown in program order: 1–6; they are, however, issued in 1 clock cycle! The notation `pi` is used to refer to a physical register; the contents of that register at any point is determined by the renaming map. For simplicity, we assume that the physical registers holding the architectural registers `x1`, `x2`, and x`3` are initially `p1`, `p2`, and `p3` (they could be any physical register). The instructions are issued with physical register numbers, as shown in column four. The rename map, which appears in the last column, shows how the map would change if the instructions were issued sequentially. The difficulty is that all this renaming and replacement of architectural registers by physical renaming registers happens effectively in 1 cycle, not sequentially. The issue logic must find all the dependences and "rewrite" the instruction in parallel.

H&P CA:A QA (6th. Ed.)

Dynamically Scheduled Multiple-issue Superscalar: Practical Limitations and Trends

- **Multiple branch predictions in the same clock cycle**
  - **Even more number of instructions fetched per clock cycle!**
  - **Extremely difficult to implement (no current product)**
- **Hardware-based Translation of CISC Instructions to RISC *Micro-operation stream***
  - **Compiler generates code targeting CISC ISA**
  - **These are translated by a HW layer during runtime**
  - **Opaque to the OS or application program**
  - **Well-suited for Dynamically Scheduled Speculative Superscalar Processors**
  - ***Instruction bundle* can be formed even by selecting microoperations corresponding to different CISC instructions!**
  - **Sometimes, selected pairs of micro-ops can be *fused* to *macro-ops* (“”), to be executed as a single operation (e.g. compare followed by branch)**
- **Focus has shifted to multi-core processor design from the early-2000s**
  - **To exploit *thread-level parallelism***
  - **Complication of speculative multiple-issue superscalar processor design**
  - **Rising power consumption => raw increase in clock frequency has fallen out of favor**

Please study the Intel Core i7 microarchitecture described in book!

Thank you