Dynamic Scheduling

This Set

- Scheduling and Dynamic Execution Definitions
  From various parts of Chapter 4.
- Description of Two Dynamic Scheduling Methods

  Not yet complete.

  (Material below may repeat material above.)
- Tomasulo's Algorithm Basics
  Section 4.2
- Reorder Buffer and Tomasulo's Algorithm
  Sections 4.2 and 4.8 plus non-text material.

  Non-text material.
- Sample Problems

Dynamic Scheduling

Scheduling:

Organizing instructions to improve execution efficiency.

Static Scheduling:

Organizing of instructions by compiler or programmer to improve execution efficiency.

Statically Scheduled Processor:

A processor that starts instructions in program order. It achieves better performance on code that had been statically scheduled. Processors covered in class up to this point were statically scheduled.

Dynamic Scheduling: [processor implementation]

A processor that allows instructions to start execution . . .

. . . even if preceding instructions are waiting for operands.

Static scheduling advantage: time and processing power available to scheduler (part of compiler).

Dynamic scheduling advantage: can execute instructions after loads that miss the cache (they will take a long time to complete). (Compiler cannot often predict load misses.) Can make up for bad or inappropriate (targeted wrong implementation) static scheduling.

Scheduling Examples

Unscheduled Code

```
add.s  f0, f1, f2
sub.s f3, f0, f4
mul.s f5, f6, f7
lwc1  f8, 0(r1)
addi  r1, r1, 8
ori   r2, r2, 1
```

Unscheduled Code on Statically Scheduled (HP Chapter-3) MIPS

```
Cycle:                 0   1   2   3   4   5   6   7   8   9   10  11
 add.s  f0, f1, f2    IF  ID  A0  A1  A2  A3  WF
 sub.s  f3, f0, f4        IF  ID  ---------->  A0  A1  A2  A3  WF
 mul.s  f5, f6, f7            IF  ---------->  ID  M0  M1  M2  M3  M4  M5  M6  WF
 lwc1   f8, 0(r1)                             IF  ID  ->  EX  MEM WF
 addi  r1, r1, 8                                  IF  ->  ID  EX  MEM WB
 ori   r2, r2, 1                                          IF  ID  EX  MEM WB
```

Execution has four stall cycles.

Statically Scheduled Code on Statically Scheduled MIPS Implementation

Instructions reordered by compiler or programmer to remove stalls.

```
Cycle:               0   1   2   3   4   5   6   7   8   9   10  11
 add.s  f0, f1, f2   IF  ID  A0  A1  A2  A3  WF
 lwc1   f8, 0(r1)        IF  ID  EX  MEM WF
 mul.s f5, f6, f4            IF  ID  M0  M1  M2  M3  M4  M5  M6  WF
 addi  r1, r1, 8                 IF  ID  EX  MEM WB
 sub.s  f3, f0, f4                   IF  ID  A0  A1  A2  A3  WF
 ori    r2, r2, 1                        IF  ID  EX  MEM WB
```

Execution has zero stall cycles.

Execution of unscheduled code on dynamically scheduled processor:

Cycle:	0	1	2	3	4	5	6	7	8	9	10	11			
`add.s  f0, f1, f2`	IF	ID	Q	RR	A0	A1	A2	A3	WC						
`sub.s  f3, f0, f4`		IF	ID	Q				RR	A0	A1	A2	A3	WC		
`mul.s f5, f6, f7`			IF	ID	Q	RR	M0	M1	M2	M3	M4	M5	M6	WC	
`lwc1  f8, 0(r1)`				IF	ID	Q	RR	EA	MEM	WB					C
`addi  r1, r1, 8`					IF	ID	Q		RR	EX	WB				C
`ori  r2, r2, 1`						IF	ID	Q		RR	EX	WB			C

Processor delays `sub.s` until `f0` is available.

Note that instructions *start* out of order (`mul.s` before `sub.s`) . . .

. . . this is called *out-of-order execution*.

(In the statically scheduled (Chapter-3) implementations FP instructions started in order and completed out of order.)

`Q`, `RR`, `WC`, and `C` are new stages.

Dynamic Scheduling Overview

Dynamic Scheduling with Register Renaming

1. After decoding instruction *rename* registers:

   Assign a temporary name to destination register. (E.g., call `r2` `pr9`.)

   Look up temporary names, in *ID register map*, for source registers.

2. Move instruction out of `ID`.

3. Execute instruction when source operands available.

4. Fetch and decode continue even if instructions are waiting to execute.

Dynamic Scheduling without Register Renaming

1. Decode instruction.
2. Stall if there are WAR or WAW hazards.
3. Move instruction out of `ID`.
4. Execute instruction when source operands available.
5. Fetch and decode continue even if instructions are waiting to execute.

Most systems covered here do perform register renaming.

Dynamic Scheduling Overview

What a dynamically scheduled processor does:

Provide storage for instructions waiting for operands.

Detect when operands for waiting instructions become available.

These will avoid stalls due to true dependencies.

Assign a new name to a register each time it is written . . .
. . . and use those names for source operands.

This will avoid stalls due to name dependencies.

Dynamic Scheduling Terminology

Issue: [an instruction]
Assignment of an instruction to a reorder buffer entry.

Initiate Execution: a.k.a., *issue*, *dispatch*
Movement of an instruction into an execution unit.

Complete: [Execution]
Movement of an instruction out of an execution unit with the result computed.

Commit: (a.k.a. *retire*, *graduate*)
Irreversibly write an instruction's results to a register or memory.

In Flight:
The state of an instruction after being issued but before being committed.

(Definitions will be illustrated in reorder buffer example.)

Dynamic Scheduling Methods

Three main methods described, differ in what temporary register name refers to:

Reorder buffer entry number.

Reservation station number.

Physical register number.

Common Features:

Use of *reorder buffer* for exception and misprediction recovery [1].

Use of *register map* to translate between architected (*e.g.*, `r1`, `f10`) register name and temporary name.

Common data bus used to broadcast instruction results.

[1] In some 20th century homeworks the reorder buffer is omitted.

Method 1: Reorder Buffer Entry Naming

Characteristics

Fastest and simplest method.

Major Parts (N.B.: Parts used differently with other methods.)

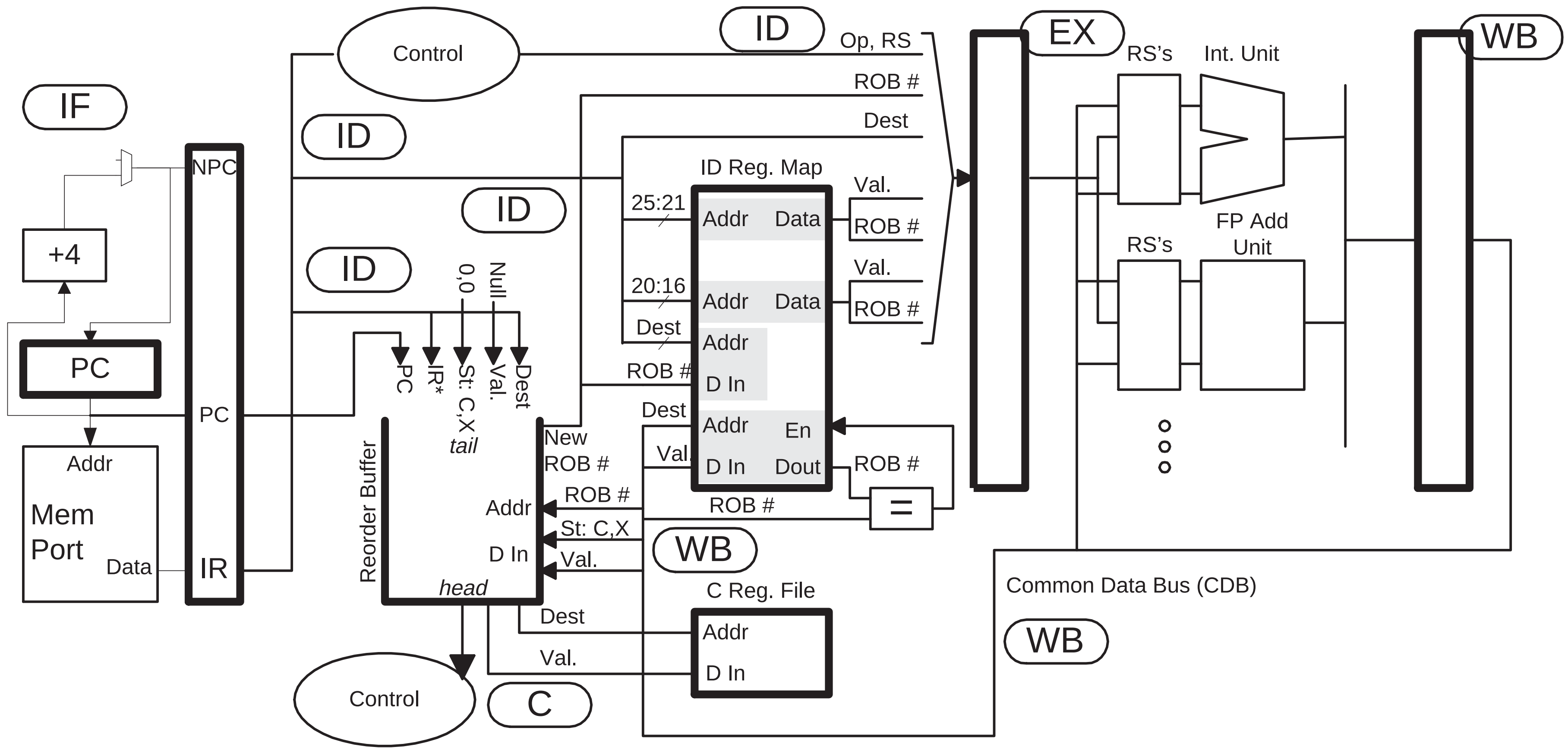
Reorder Buffer (ROB): a.k.a. *Active List*
A list, in order, of in-flight instructions.

ID Register Map:
Used in place of register file. Provides value or ROB entry # for each register.

Commit Register File:
Holds committed register values, used for recovery.

Reservation Station:
A buffer where instructions wait to execute.

(Method 1) Hardware



Stage name shown next to hardware used in that stage.

Branch hardware, immediates, load/store hardware not shown.

Connections used for recovery not shown.

Hardware in ID stage spans several stages in real systems.

(Method 1) Reorder Buffer

Each entry holds information on an instruction:

PC	IR	Status	Value	Dest..

PC: Program counter (address) of instruction.

IR: The decoded instruction. (Not used here.)

Status: Whether completed, and whether raised an exception.

Value: The result produced (if applicable).

Dest.: The destination register to be written.

An instruction "uses" the reorder buffer three times.

(Method 1) ID Register Map

Register Map

Used in ID and WB stage.

Indexed using architected register number.

When an architected register number placed at `Addr` input . . .
. . . provides the latest value or . . .
. . . the ROB entry # of instruction that will produce latest value.

Has four ports:

Two for reading source operands (during ID).

One for writing the new ROB # of the destination (during ID).

One for reading ROB # and (if necessary) writing results (during WB).

(Method 1) Commit Register File

Commit Register File

Works like register file in statically scheduled (Chapter-3) implementation.

Values written when instructions commit.

Used to recover from an exception or misprediction.

(Method 1) Reservation Station

Reservation Station:
A buffer where instructions wait to execute.

Each functional unit has a set of reservation stations.

In ID an instruction is assigned to a RS based on opcode.

Reservation Station Entry:

Op	ROB#	Dest	Val1	ROB#1	Val2	ROB#2

Op: Operation (May not be same as opcode, but specifies same operation.)

ROB#: Reorder buffer entry holding instruction (name of result).

Dest: Destination register number.

Val1,ROB#1: Value of $\langle$**rs1**$\rangle$ (operand 1) or ROB entry of instruction that will produce it.

Val2,ROB#2: Value of $\langle$**rs2**$\rangle$ or ROB entry of instruction that will produce it.

(Method 1) Common Data Bus

Common Data Bus (CDB):
A bus connecting functional units to other parts of the processor, used to broadcast results.

Data on CDB:

Dest	ROB#	Status	Value

Dest: Destination register.

ROB #: Reorder buffer entry number. (Temporary name of dest.)

Status: Happy ending, or an exception?

Value: The result. Can't forget that.

(Method 1) Operation

Stages in execution of an instruction.

IF: Same as Chapter 3. (Will change later.)

ID:

Initialize new reorder buffer entry.

Reorder buffer provides the new ROB #.

`Val` can be set to anything, `Status` bits: complete, 0; exception, 0.

Controller chooses reservation station.

Read operands from register map.

Result of read is either value or ROB #.

Write ROB# (new name for destination) to register map.

RS/EX:

If both operands found in register map, start execution, otherwise wait in reservation station.

If waiting in reservation station:

"Listen" to CDB for ROB # of missing operands ...
... when "heard" copy value into RS entry.

When both values available, execution can start.

WB:

When functional unit completes execution it tries to get control of CDB (contending with other units).

When it gets control it places result and other information on the bus . . .

. . . instructions waiting in RS may copy the result (if they need it) . . .

. . . the result may be written into the register map . . .

. . . the status (hopefully complete, but there could have been an exception) is written into the reorder buffer.

An instruction in WB *reads* the ROB # for the architected destination register from the ID map . . .

. . . if it matches the instruction's own ROB # the result is written.

C (Commit):

When instruction reaches the head of the ROB . . .
. . . the controller checks the status.

If execution complete and unexceptional:

Value written into register file.

If execution complete but encountered an exception:

Recovery is initiated . . .
. . . All instructions in reorder buffer squashed, . . .
. . . the register file is copied to the register map, and . . .
. . . exception handler address loaded into PC.

(Method 1) Comments

Method for handling loads and stores covered later.

Details, such as handling of immediates and branches omitted.

If it's the 21st century, ask for more examples.

A register value can be in several places at once:

- The register map, the reorder buffer, the register file, and a reservation station.
- This uses alot of storage and wires (which is expensive to implement) but has two advantages:
  - Relatively simple to explain.
  - Fast because values are stored at functional units (in RS).

Method 2: Reservation Station Entry Naming

Differences with method 1:

Reservation station number, rather than ROB entry used to identify operands.

Reservation station held by an instruction *until it completes execution* . . .
. . . to avoid name duplication.

For example, `add.s` below keeps RS 0 and `sub.s` is assigned RS 1. This way when `add.s` finishes and writes RS 0 on the CDB to identify the result we can be sure no other instruction is using RS 0 for the result (as `sub.s` might have).

```
! Cycle                 0    1    2    3    4    5    6    7    8    9   10
 add.s  f0, f1, f2      IF   ID 0:A0 0:A1 0:A2 0:A3   WC
 sub.s  f0, f0, f4           IF   ID 1:RS 1:RS 1:RS 1:A0 1:A1 1:A2 1:A3   WC
 mul.s  f5, f0, f6                IF   ID 2:RS 2:RS 2:RS 2:RS 2:RS 2:RS 2:M1...
```

No diagram for method 2 (since very similar to method 1).

Method 3: Physical Register Naming

Difference with method 1:

A single register file, called the *physical register file* used to store register values.

Instructions wait in an *instruction queue* for execution. (No reservation stations.)

A *scheduler* chooses instructions from queue to execute . . .
. . . when chosen they read registers and *in the next cycle*. . .
. . . move to an execution unit.

Method 3: Physical Register Naming, Continued

Two register maps are used:

ID Register Map:

Provides physical register numbers for architected registers. Updated in ID, so physical registers might not yet have values.

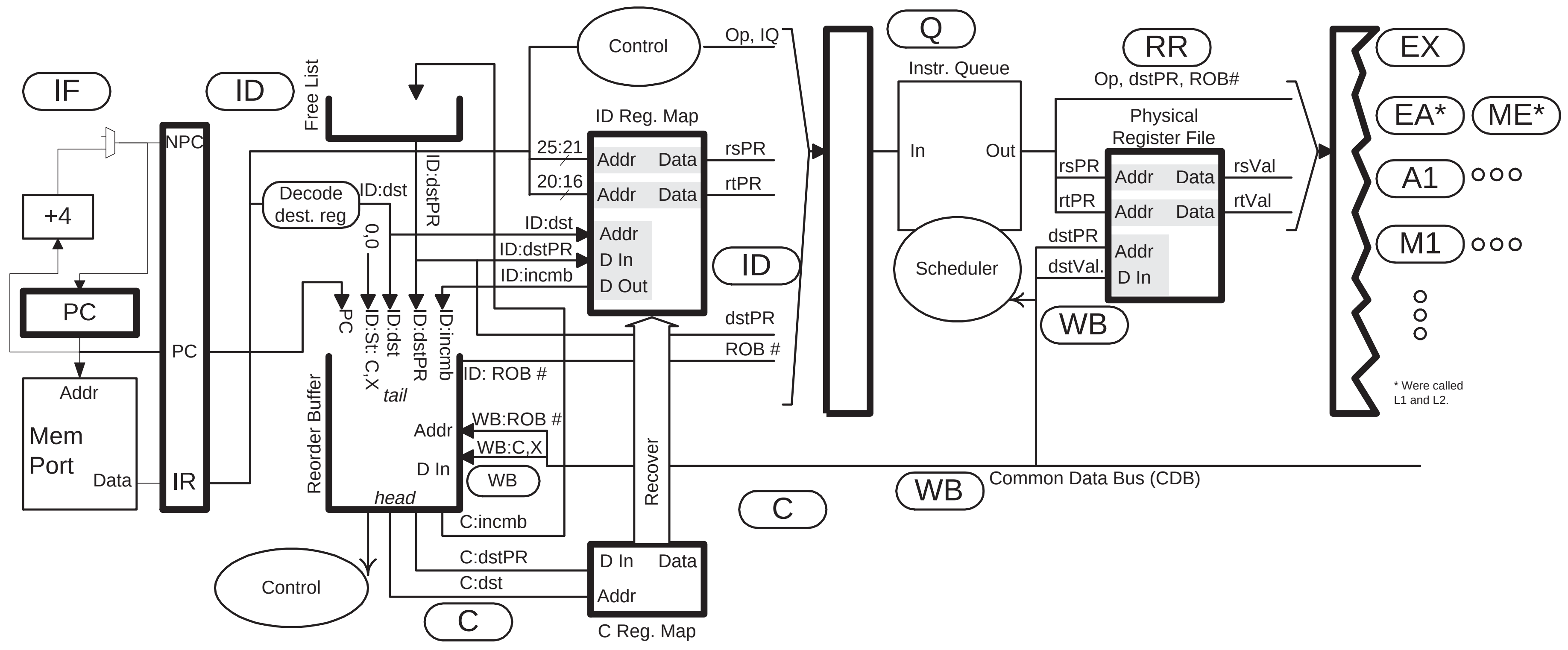
Commit (C) Register Map:

Provides physical register numbers for architected registers. Updated at commit, so physical register holds a valid value. (Used for recovery.)

Free List:

Holds list of unused physical registers.

(Method 3) Hardware



Branch prediction hardware not shown.

Immediate handling and other small details not shown.

Most recovery connections not shown.

In real systems ID and Q span many more stages.

Loads and Stores in Dynamically Scheduled Processors

Load/Store Unit (LSU)

Loads and stores done in two steps:

`EA` (was called `L1`): Address computation, and

`ME` (was called `L2`): Memory read or write.

`ME` may encounter a *cache miss* (data far away).

Cache used with load/store unit can be *blocking* or *non-blocking*.

If cache is *blocking* must wait for data to arrive before attempting other loads or stores.

For example,

```
! Example: 0(r4) in cache, but 0(r2) not in cache.
! Cycle          0   1   2   3   4   5                  10  11  12  13  14
lw  r1, 0(r2)    IF  ID  Q   RR  EA  ME  ------...  --> WB
lh  r3, 0(r4)        IF  ID  Q   RR  EA  ------...  --> ME  WB
add r10, r10, r1         IF  ID  Q                      RR  EX  WB
sub r11, r11, r3             IF  ID  Q                      RR  EX  WB
```

Because cache is blocking miss by `lw` forces `lh` (and `sub`) to wait, even though data cached.

If cache is *non-blocking* can attempt other loads and stores while handling miss:

With a non-blocking cache, instruction would not wait in `ME` during cache miss.

For example,

```
! Example: 0(r4) in cache, but 0(r2) not in cache.
! Cycle          0   1   2   3   4   5   6   7   8 ...  12  13  14
lw  r1, 0(r2)    IF  ID  Q   RR  EA  ME                 ME  WB
lh  r3, 0(r4)        IF  ID  Q   RR  EA  ME  WB
add r10, r10, r1         IF  ID  Q                      RR  EX  WB
sub r11, r11, r3             IF  ID  Q   RR  EX  WB
```

Because cache is non-blocking miss by `lw` does not delay `lh` (or `sub`).

Load and Store Dependencies

Loads and stores have dependencies too.

Consider:

```
! At cycle 5 we know 0(r1) != 0(r5), but the processor is not omniscient.
! Cycle           0   1   2   3   4   5   6   7   ... 12  13  14  15  16
lw  r1, 0(r2)     IF  ID  Q   RR  EA  ME  -------...  ME  WB
sw  0(r1),r3          IF  ID  Q                       RR  EA  ME  WB
lw  r4, 0(r5)             IF  ID  Q   RR  EA  ME ...              ME  WB
add r5, r4, r5                IF  ID  Q                           RR  EX  WB
```

`sw` had to wait because of the true dependency through `r1`.

Since `r1` not available at cycle 7 the second `lw` has to wait because of ...

... a *possible* address dependency: `0(r1)`$\overset{?}{=}$`0(r5)`.

The situation is different when the store *address* is available:

Consider:

```
! At cycle 5 we know 0(r3) != 0(r5), and the processor does too.
! Cycle          0   1   2   3   4   5   6   7   8   9   10  11
lw  r1, 0(r2)    IF  ID  Q   RR  EA  ME  -------...  ME  WB
sw  0(r3),r1         IF  ID  Q   RR  EA                  ME  WB
lw  r4, 0(r5)            IF  ID  Q   RR  EA  ME  WB
add r5, r4, r5               IF  ID  Q       RR  EX  WB
```

Here, the second `lw` uses a different address than the `sw`, so it doesn't have to wait.

Load/Store Units for Non-Blocking Caches

LSU maintains a queue of instructions in program order.

It keeps track of which instructions are waiting for addresses.

Store instructions write the cache when they commit.

Load instructions can read (*bypass*) data from store instructions in the queue.

The memory (`ME`) part of a load operation can proceed if. . .
. . . its address is available . . .
. . . addresses for *all* preceding queued store instructions are available . . .
. . . the load address does not match any preceding store instructions or. . .
. . . the address does match a store and the (latest) store data is available.

Editorial Comment

The slides that follow contain material that has not yet been integrated into the material above, and so it may seem repetitive or out of place.

Dynamic Scheduling Methods

Scoreboard

Avoids stalls due to true dependencies.

Covered in text section 4.2, but not in class.

Tomasulo's Algorithm

Avoids stalls due to name dependencies . . .
. . . by assigning a new name to (renaming) each destination register.

Avoids stalls due to true dependencies . . .
. . . by holding instructions (in reservation stations) waiting for operands.

Three variations covered in class.

Widely used in real systems.

Tomasulo's Algorithm Basics

Implementation methods vary, a simple system described.

Reservation Station (RS)

Buffer area where instructions wait for operands and an execution unit.

Each functional unit has several reservation stations.

Common Data Bus (CDB)

A bus connecting functional unit output to . . .
. . . register file (in some cases) . . .
. . . reservation stations . . .
. . . other devices awaiting instruction completion.

CDB used to pass results from functional unit to . . .
. . . waiting instructions . . .
. . . and (in some cases indirectly) the register file.

Tomasulo's Algorithm Basics

When an instruction is in ID (during issue):

A reservation station is assigned to the instruction.

A temporary name is chosen for destination register.

Following instructions refer to that temporary name.

When an instruction completes execution:

Result broadcast on CDB along with temporary name and other information.

Result read from CDB by instructions in RS that need it.

Depending on variation result is...
... written to register file (if it's the latest value) ...
... or written to other storage area before reaching register file.

Execution With Reservation Stations

Pipeline Execution Diagram Notation

Instruction in reservation station x indicated by: x:RS.

Instruction using RS 3 in stage 4 of multiply: 3:M4.

Timing

Option 1: CDB to execution unit and RS.

Waiting instruction starts while dependent instruction in WB.

Option 2: CDB to RS, RS to execution unit.

Waiting instruction starts cycle after dependent instruction in WB.

Example:

Reservation station numbers: fp add unit, 0-3; fp mult unit, 4-5.

Timing uses option 1, reservation station numbers will be used for temporary names.

Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13
`multf f0, f1, f2`	IF	ID	4:M1	4:M2	4:M3	4:M4	4:M5	4:M6	4:M7	4:WB				
`subf f3, f0, f4`		IF	ID	0:RS	0:RS	0:RS	0:RS	0:RS	0:RS	0:A1	0:A2	0:A3	0:A4	0:WB
`addf f0, f5, f6`			IF	ID	1:A1	1:A2	1:A3	1:A4	1:WB					
`ltf f0, f7`				IF	ID	2:RS	2:RS	2:RS	2:A1	2:A2	2:A3	2:A4	2:WB	

In cycle 1 `multf` assigned RS 4 and it reads values for `f1` and `f2`. RS 4 is the temporary name given to its result, `f0`.

In cycle 2 `subf` assigned RS 0. It reads the value of `f4` and the reservation station that will produce `f0`, RS 4.

In cycle 3 `subf` sits in RS 0 waiting for `multf` to finish to get `f0`, a.k.a., RS 4.

Meanwhile, in ID, `addf` is assigned RS 1, `f0` will now be known as RS 1. Values for `f5` and `f6` are read.

In cycle 4 `ltf` assigned RS 2. It reads a value for `f7` but it will have to wait for RS 1 to finish to get `f0`. (`f0` and `f7` are source operands of `ltf`, the destination is the floating-point condition code register).

Example, continued.

Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13
multf f0, f1, f2	IF	ID	4:M1	4:M2	4:M3	4:M4	4:M5	4:M6	4:M7	4:WB				
subf f3, f0, f4		IF	ID	0:RS	0:RS	0:RS	0:RS	0:RS	0:RS	0:A1	0:A2	0:A3	0:A4	0:WB
addf f0, f5, f6			IF	ID	1:A1	1:A2	1:A3	1:A4	1:WB					
ltf f0, f7				IF	ID	2:RS	2:RS	2:RS	2:A1	2:A2	2:A3	2:A4	2:WB	

In cycle 8 `addf` writes the common data bus, `ltf` copies the result and starts execution.

In cycle 8 the register file is updated since RS 1 is the latest name of `f0`. (The register file would not be updated in this cycle if a reorder buffer were being used. See next example.)

In cycle 9 `multf` writes the CDB, `subf` copies the result.

In cycle 9 the register file is **not** updated since RS 4 is an outdated name for `f0`.

Example:

Reservation station numbers: fp add unit, 0-1; fp mult unit, 4-5, 6-9 integer unit.

Add unit has one less reservation station than in last example.

Timing uses option 2.

Reservation stations will be used for temporary names.

```
Cycle              0   1     2    3    4    5    6    7    8    9    10   11   12   13   14
 multf f0, f1, f2  IF  ID  4:M1 4:M2 4:M3 4:M4 4:M5 4:M6 4:M7 4:WB
 subf  f3, f0, f4      IF    ID 0:RS 0:RS 0:RS 0:RS 0:RS 0:RS 0:RS 0:A1 0:A2 0:A3 0:A4 0:WB
 addf  f0, f5, f6            IF   ID 1:A1 1:A2 1:A3 1:A4 1:WB
 ltf   f0, f7                     IF   ID ------------------> 1:A1 1:A2 1:A3 1:A4 1:WB
 xor   r1, r2, r3                      IF ------------------>   ID 6:EX 6:WB
```

Because of CDB timing option 2, `subf` waits an extra cycle to start, during cycle 9.

Because there are not enough add reservation stations, decoding stalls at cycle 5.

Dynamic Scheduling with Tomasulo's Algorithm Variations

Three variations, differ in what "temporary name" refers to:

- Reservation Stations Only

  Temporary name is reservation station number of instruction producing result. (Used in examples above.)

- Reorder Buffer and Reservation Stations

  Temporary name is *reorder buffer* entry number of instruction producing result.

- Reorder Buffer, Reservation Stations, Physical (Rename) Registers

  Temporary name is name of a special *rename register*.

Second and third variations commonly used.

Reorder Buffer

Reorder Buffer

A buffer holding information about instructions, kept in program order.

Each instruction occupies a *reorder buffer entry*.

Each entry has a unique number which can be used to identify the instruction.

Reorder Buffer Use

In ID reorder buffer entry created for instruction.

Entry updated during instruction execution.

Entry removed if . . .
. . . it is the oldest entry . . .
. . . and the instruction has completed execution.

When an entry is removed . . .
. . . the register file is written if the instruction writes a register . . .
. . . memory is written if the instruction writes memory.

When an entry is removed the instruction is said to *commit*.

Hardware limits number of instructions that can commit per cycle . . .
. . . usually the same as the number that can issue per cycle (1 so far).

Commit shown in execution diagram with a WC or $\boxed{\text{WB}}$ if it occurs during writeback or C if it occurs later.

Removal of item from reorder buffer sometimes called *retirement*.

Example:

Reservation station numbers: fp add unit, 0-3; fp mult unit, 4-5.

Timing uses option 1.

Reorder buffer entry numbers will be used for temporary names. (Entry numbers not shown in diagram.)

```
Cycle              0   1     2    3    4    5    6    7    8    9    10   11   12   13  14 15
 multf f0, f0, f2  IF  ID    M1   M2   M3   M4   M5   M6   M7   WC
 subf  f3, f0, f4      IF    ID 0:RS 0:RS 0:RS 0:RS 0:RS 0:RS   A1   A2   A3   A4   WC
 addf  f0, f5, f6            IF   ID   A1   A2   A3   A4   WB                            C
 ltf   f0, f7                     IF   ID 2:RS 2:RS 2:RS   A1   A2   A3   A4   WB          C
```

RS assignment same as earlier examples however RS freed when execution starts. Note commit symbols in diagram.

Table showing time of events during instruction execution:

Instr	Issue	Initiate Ex.	Complete Ex.	Commit
`multf`	1	2	8	9
`subf`	2	9	12	13
`addf`	3	4	7	14
`ltf`	4	8	11	15

Reorder Buffer and Exceptions

Precise exceptions easily provided with a reorder buffer.

Reorder buffer entry has an exception bit.

Bit is tested when instruction reaches head of buffer (is oldest in buffer).

If bit is set reorder buffer cleared and a trap instruction inserted in pipeline.

Because bit tested when faulting instruction reaches buffer head . . .
. . . all preceding instructions have executed and their registers and memory written.

Because buffer cleared, none of the following instructions have written registers or memory.

Therefore, exceptions are precise.

Register Mapping

Register Mapping (noun)

The temporary name (RS, reorder buffer entry, or rename register) of an architecturally visible register.

Register Mapping (verb)

The process of finding the temporary name of an architecturally visible register.

Method of Register Mapping (useful in all variations)

Maintain a table indexed by architecturally visible register number ...

... the table can be part of the register file itself ...

... or use a separate memory device (easier to implement).

The table provides the latest temporary name, if any.

(If latest value is in register file, there is no temporary name.)

Table called a *register map*.

Register Mapping and Cancelled Instructions

Initially all registers map to the architecturally visible ones.

As instructions are issued the register map is updated with temporary names.

If an in-flight instruction is cancelled, the register map file may become invalid.

To safely cancel an instruction (*e.g.*, for an exception) when using a reorder buffer:

Wait for instruction to reach head of reorder buffer.

At this point the register file has been updated for all preceding instructions.

Cancel remaining instructions.

Reset the map file (so that all registers map to the architecturally visible ones).

Sample Problems

Dynamic Scheduling

EE 4720 terminology notes:

Before 1999, dynamic scheduling called dynamic issue in course materials.

Rename registers were not used in problems before 1999. Do not confuse *register renaming* (assignment of a temporary name to a destination register) with *rename registers* (a set of additional registers to hold results until an instruction commits).

1998 Homework 4, Problems 4 and 5.

1997 Final Exam, Problems 2a and 2b.

See also multiple-issue sample problems, at end of lecture notes sets 11 and 12.

Reorder Buffer

EE 4720 terminology note: before 1999 the term *retire* is used instead of *commit*.

1998 Homework 6, Problem 1. (Includes later material.)

1998 Final Exam, Problem 2. (Includes later material.)