
16.482 / 16.561

Computer Architecture and
Design

Instructor: Dr. Michael Geiger
Spring 2014

Lecture 5:
Dynamic scheduling

Lecture outline

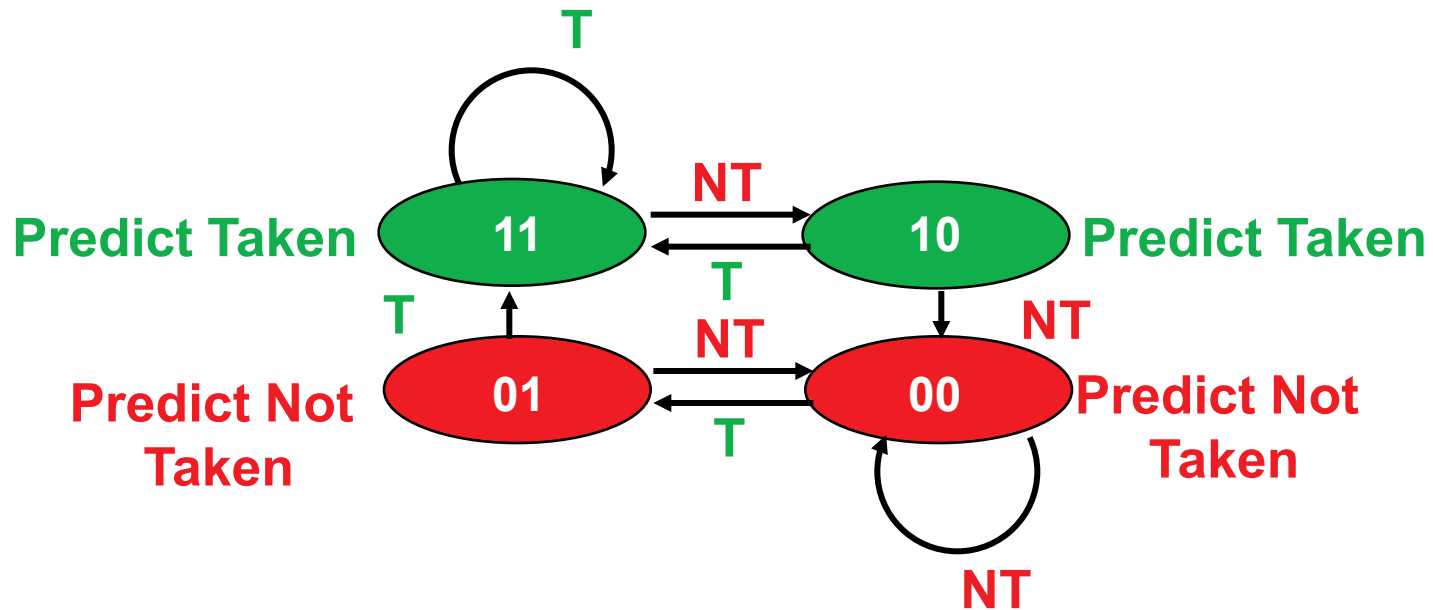
- Announcements/reminders
 - HW 3 due today
 - HW 4 to be posted; due 3/6
- Review
 - Dynamic branch prediction
- Today's lecture
 - Dependences and hazards (review)

Review: Dynamic Branch Prediction

- Want to avoid branch delays
- **Dynamic branch predictors**: hardware to predict branch **outcome** (T/NT) in 1 cycle
 - Use branch history to determine predictions
 - Doesn't calculate target
- **Branch history table**: basic predictor
 - Which line of table should we use?
 - Use appropriate bits of PC to choose BHT entry
 - # index bits = $\log_2(\# \text{ BHT entries})$
 - What's prediction?
 - How does actual outcome affect next prediction?

Review: BHT

- Solution: 2-bit scheme where change prediction only if get misprediction twice



- Red: “stop” (branch not taken)
- Green: “go” (branch taken)

Review: Correlated predictors, BTB

- Correlated branch predictors
 - Track both individual branches and overall program behavior (**global history**)
 - Makes some branches easier to predict
 - To make a prediction
 - Branch address chooses row
 - Global history chooses column
 - Once entry chosen, make prediction in same way as basic BHT (11/10 → predict T, 00/01 → predict NT)
- Branch target buffers
 - Save previously calculated branch targets
 - Use branch address to do fully associative search

Data Dependence and Hazards

- Instr_J is **data dependent** (aka **true dependence**) on Instr_I if:
 - Instr_J tries to read operand before Instr_I writes it
 - ⤵ I: **add \$1, \$2, \$3**
 - ⤵ J: **sub \$4, \$1, \$3**
 - or Instr_J is data dependent on Instr_K which is dependent on Instr_I
- If two instructions are data dependent, they cannot execute simultaneously or be completely overlapped
- If data dependence caused a hazard in pipeline, called a **Read After Write (RAW) hazard**

ILP and Data Dependences, Hazards

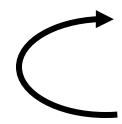
- HW/SW must preserve **program order**
 - Determined by source code
 - Implies data flow and calculation order
 - Therefore, dependences are a property of programs
 - Limits ILP
- Dependence indicates potential hazard
 - Actual hazard and length of any stall is property of the pipeline
- HW/SW goal: exploit parallelism by preserving program order only where it affects the outcome of the program

Name dependences

- **Name dependence:** 2 instructions use same register or memory location, but **no data flow** between instructions associated with that name
- Name dependences only cause problems if program order is changed
 - In-order program suffers no hazards from these dependences
- Can be resolved through register renaming
 - Will revisit with dynamic scheduling

Name Dependence #1: Anti-dependence

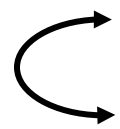
- **Anti-dependence:** Instr_j writes operand before Instr_i reads it

 I: sub \$4, \$1, \$3
J: add \$1, \$2, \$3
K: mul \$6, \$1, \$7

- If anti-dependence causes a hazard in the pipeline, called a **Write After Read (WAR) hazard**

Name Dependence #2: Output dependence

- **Output dependence:** Instr_j writes operand before Instr_i writes it.

 I: sub \$1, \$4, \$3
J: add \$1, \$2, \$3
K: mul \$6, \$1, \$7

- If output dependence causes a hazard in the pipeline, called a **Write After Write (WAW)** hazard

Loop-carried dependences

- Easy to identify dependences in basic blocks
- Trickier across loop iterations

- Example:

```
L:  add  $t0, $t1, $t2
     lw  $t2, 0($t0)
     cmp $t2, $zero
     bne L
```

- `$t2` from `lw` used in next loop iteration
- **Loop-carried dependence:** dependence in which value from one iteration used in another

Dependence example

- Given the code below

```
Loop:      ADD    $1, $2, $3
I0:        ADD    $3, $1, $5
I1:        LW     $6, 0($3)
I2:        LW     $7, 4($3)
I3:        SUB    $8, $7, $6
I4:        DIV   $7, $8, $1
I5:        ADDI   $4, $4, 1
I6:        SW     $8, 0($3)
I7:        SLTI   $9, $4, 50
I8:        BNEZ   $9, Loop
```

- List the data dependences
 - Assuming a 5-stage pipeline with no forwarding, which of these would cause RAW hazards?
- List the anti-dependences
- List the output dependences

Dependence example solution

- Data dependences (RAW hazards underlined)

\$1: Loop → I0

\$1: Loop → I4

\$3: I0 → I1

\$3: I0 → I2

\$3: I0 → I6

\$6: I1 → I3

\$7: I2 → I3

\$8: I3 → I4

\$8: I3 → I6

\$4: I5 → I7

\$9: I7 → I8

Dependence example solution (cont.)

- **Anti-dependences**

\$3: Loop \rightarrow I0

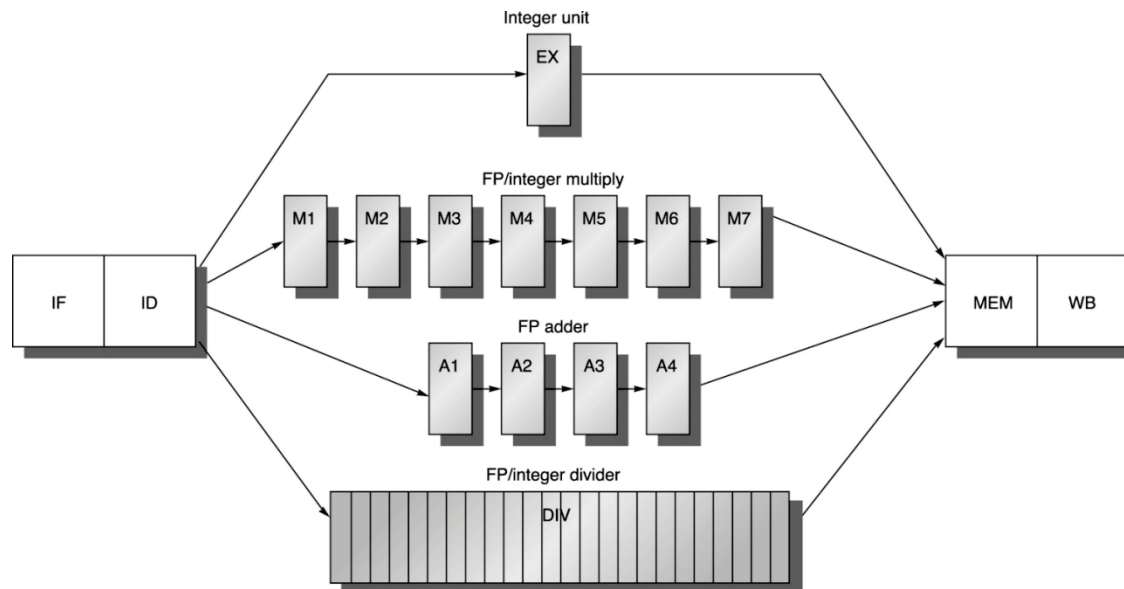
\$7: I3 \rightarrow I4

- **Output dependences**

\$7: I2 \rightarrow I4

Realistic pipeline

- A 5-stage pipeline is unrealistic for a modern microprocessor
 - Floating point (FP) ops take much more time than integer ops
 - Solution: Pipelined execution units
 - Allow integer ops (ADD, SUB, etc.) to finish in 1 cycle
 - Allow multiple FP ops of a particular type to execute at once
 - Example: in pipeline below, can have up to 4 ADD.D instructions at once
 - May also pipeline memory accesses (not shown below)



MIPS floating point

- 32 SP floating point registers (F0-F31)
- Registers paired for double precision ops
 - For example, in a double-precision add, “F0” refers to the register pair F0/F1
- Arithmetic instructions similar to integer
 - “.s” or “.d” at end of instruction for single/double
 - add.d, sub.d, mult.d, div.d
- Data transfer
 - Load: L.S / L.D
 - Store: S.S / S.D

Latency and stalls

- For our purposes, an instruction's **latency** is equal to the **number of pipeline stages in which that instruction does useful work**
- In the realistic pipeline slide:
 - Integer ops have a 1 cycle latency (EX)
 - Multiply ops have a 7 cycle latency (M1-M7)
 - FP adds have a 4 cycle latency (A1-A4)
 - Divide ops have a 24 cycle latency (D1-D24)
 - Memory ops have a $1+1 = 2$ cycle latency
 - Address calculation in EX, memory access in MEM

Determining stalls

- Most of the time, assuming forwarding:
(# cycles between dependent instructions) =
(latency of producing instruction – 1)
 - If no instructions between those dependent instructions, those cycles become **stalls**
 - Note: cycle that gets stalled is the cycle in which value is used

Case #1: ALU to ALU

■ Most common case:

- Instruction produces result during EX stage(s)
- Dependent instruction uses result in its own EX stage(s)
- Easy to see stalls = (latency – 1) here
- Note: same rule applies for ALU → load/store if ALU result is used for address calculation

	1	2	3	4	5	6	7	8	9	10
ADD.D	IF	ID	EX1	EX2	EX3	M	WB			
ADD.D		IF	ID	S	S	EX1	EX2	EX3	M	WB

Case #2: Load to ALU

- Load produces result at end of memory stage
- ALU op uses result at start of EX stage(s)
- If you consider total latency (EX + MEM) for load, stalls = (latency – 1)

	1	2	3	4	5	6	7	8	9
L.D	IF	ID	EX	M	WB				
ADD.D		IF	ID	S	EX1	EX2	EX3	M	WB

Case #3: ALU to store

- Assumes ALU result is stored into memory
- Appears only one stall is needed ...
 - What's problem?

	1	2	3	4	5	6	7	8	9	10
ADD.D	IF	ID	EX1	EX2	EX3	M	WB			
S.D		IF	ID	EX	S	M	WB			

Case #3: ALU to store (cont.)

- Structural hazard on MEM/WB stages
 - Requires additional stall
- Note that hazard shouldn't exist
 - ADD.D doesn't really use MEM stage
 - S.D doesn't really use WB stage
 - Current pipeline forces us to share hardware; smarter design will alleviate this problem and reduce stalls

	1	2	3	4	5	6	7	8	9	10
ADD.D	IF	ID	EX1	EX2	EX3	M	WB			
S.D		IF	ID	EX	S	S	M	WB		

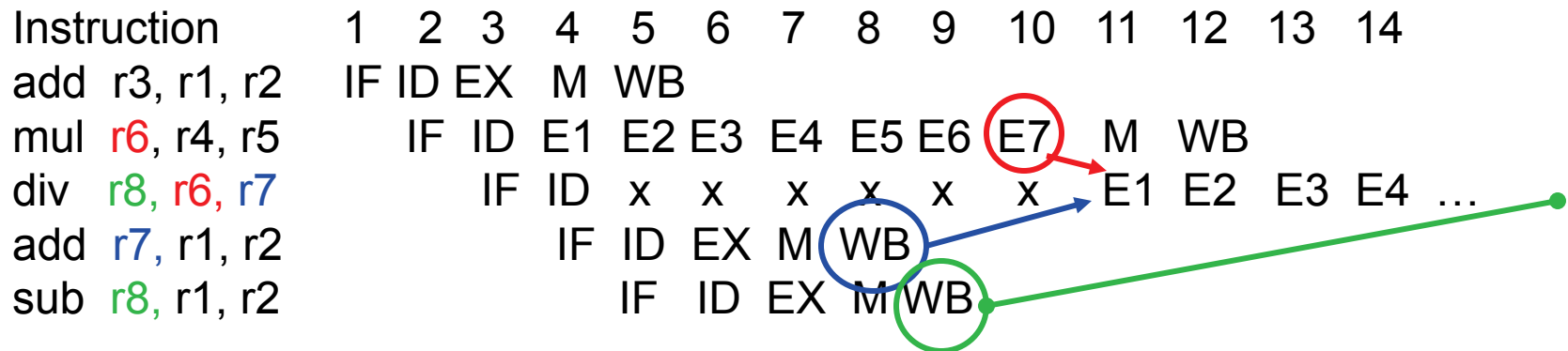
Case #4: Load to store

- The one exception to the rule
 - Value loaded from memory; stored to new location
 - Used for memory copying
 - # stalls = (memory latency – 1)
 - Forwarding from one memory stage to the next
 - 0 cycles in our examples

	1	2	3	4	5	6	7	8	9	10
L.D	IF	ID	EX	M	WB					
S.D		IF	ID	EX	M	WB				

Out-of-order execution

- Variable latencies make out-of-order execution desirable
- How do we prevent **WAR** and **WAW** hazards?
- How do we deal with variable latency?
 - Forwarding for **RAW** hazards harder



Advantages of Dynamic Scheduling

- **Dynamic scheduling:** hardware rearranges instruction execution to reduce stalls while maintaining data flow and exception behavior
- **Benefits**
 - Handles cases when dependences unknown at compile time
 - Allows processor to tolerate unpredictable delays (i.e., cache misses) by executing other code while waiting
 - Allows code that compiled for one pipeline to run efficiently on a different pipeline
 - Simplifies the compiler
- **Hardware speculation,** a technique with significant performance advantages, builds on dynamic scheduling
 - Combination of dynamic scheduling and branch prediction

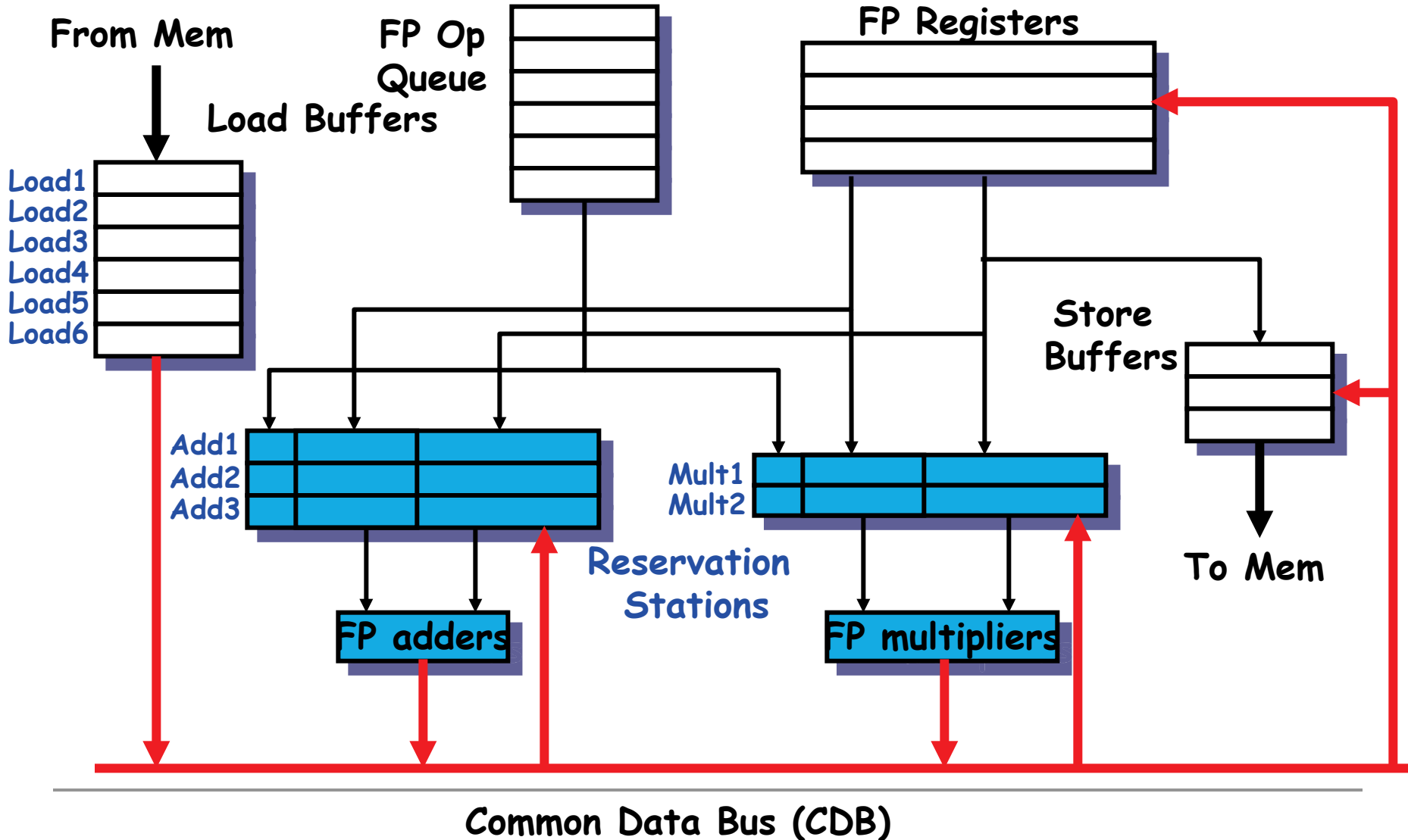
HW Schemes: Instruction Parallelism

- Key idea: Allow instructions behind stall to proceed
 - DIVD **F0**, F2, F4
 - ADDD F10, **F0**, F8
 - SUBD F12, F8, F14**
- Enables **out-of-order execution** and allows **out-of-order completion** (e.g., SUBD)
 - In a dynamically scheduled pipeline, all instructions still pass through issue stage in order (**in-order issue**)
- Note: Dynamic execution creates WAR and WAW hazards and makes exceptions harder

Tomasulo's Algorithm

- Control & buffers **distributed** with Function Units (FU)
 - FU buffers called “reservation stations”; have pending operands
- Registers in instructions replaced by values or pointers to reservation stations(RS) (register renaming)
 - Renaming avoids WAR, WAW hazards
 - More reservation stations than registers, so can do optimizations compilers can't
- Results to FU from RS, *not through registers*, over Common Data Bus that broadcasts results to all FUs
 - Avoids RAW hazards by executing an instruction only when its operands are available
- Load and Stores treated as FUs with RSs as well
- Integer instructions can go past branches (predict taken), allowing FP ops beyond basic block in FP queue

Tomasulo Organization



Reservation Station Components

- **Op**: Operation to perform in the unit (e.g., + or −)
- **V_j, V_k**: Value of Source operands
 - Store buffers has V field, result to be stored
- **Q_j, Q_k**: Reservation stations producing source registers (value to be written)
 - Note: Q_j, Q_k=0 => ready
 - Store buffers only have Q_i for RS producing result
- **A**: Address (memory operations only)
- **Busy**: Indicates reservation station or FU is busy

Implementing Register Renaming

- **Register result status table**
 - Indicates which instruction will write each register, if one exists
 - Holds name of reservation station with producing instruction
 - Blank when no pending instructions that will write that register
 - When instructions try to read register file, check this table first
 - If entry is empty, can read value from register file
 - If entry is full, read name of reservation station that holds producing instruction

F0	F2	F4	F6	F8
Load1		Add1	Mult1	

Instruction execution in Tomasulo's

- **Fetch**: place instruction into Op Queue (IF)
- **Issue**: get instruction from FP Op Queue (IS)
 - Find free reservation station (RS)
 - If RS free, check **register result status** and CDB for operands
 - If available, get operands
 - If not available, read new register name(s) and place in Qj / Qk
 - Rename result by setting appropriate field in register result status
- **Execute**: operate on operands (EX)
 - Instruction starts when both operands ready and func. unit free
 - Checks **common data bus (CDB)** while waiting
 - We allow EX to start in same cycle operand is received
 - Number of EX (and MEM) cycles depends on latency

Instruction execution in Tomasulo's

- **Memory access**: only happens if needed!
(MEM)
- **Write result**: finish execution, send result
(WB)
 - Broadcast result on CDB
 - Waiting instructions read value from CDB
 - Write to register file only if result is newest value for that register
 - Check register result status—see if RS names match
 - Assume only 1 CDB unless told otherwise
 - Potential structural hazard!
 - Oldest instruction should broadcast result first

Renaming example

- Given the following available reservation stations:
 - Add1-Add4 (ADD.D/SUB.D)
 - Mult1-Mult2 (MULT.D/DIV.D)
 - Load1-Load2 (L.D)
- Rewrite the code below with renamed registers, replacing register names with appropriate reservation stations. It may help to track the register result status for each instruction.

```
L.D    F2, 0(R1)
ADD.D  F0, F2, F6
SUB.D  F6, F0, F2
MULT.D F2, F6, F0
DIV.D  F6, F2, F6
S.D    F6, 8(R1)
```

Solution

- Assume reservation stations are assigned in order
- Resulting code

L.D **Load1**, 0 (R1)

ADD.D **Add1**, **Load1**, F6

SUB.D **Add2**, **Add1**, **Load1**

MULT.D **Mult1**, **Add2**, **Add1**

DIV.D **Mult2**, **Mult1**, **Add2**

S.D **Mult2**, 8 (R1)

Tomasulo's example

- Assume the following latencies
 - 2 cycles (1 EX, 1 MEM) for memory operations
 - 3 cycles for FP add/subtract
 - 10 cycles for FP multiply
 - 40 cycles for FP divide
- We'll look at execution of the following code (solution to be posted separately)

```
L.D      F6, 32(R2)
L.D      F2, 44(R3)
MUL.D    F0, F2, F4
SUB.D    F8, F6, F2
DIV.D    F10, F0, F6
ADD.D    F6, F8, F2
```

Dynamic loop unrolling

- Why can Tomasulo's overlap loop iterations?
 - Register renaming
 - Multiple iterations use different physical destinations for registers (dynamic loop unrolling).
 - Reservation stations
 - Permit instruction issue to advance past integer control flow operations
 - Also buffer old values of registers - totally avoiding the WAR stall

Tomasulo's advantages

- 1. Distribution of the hazard detection logic**
 - ❑ distributed reservation stations and the CDB
 - ❑ If multiple instructions waiting on single result, & each instruction has other operand, then instructions can be released simultaneously by broadcast on CDB
 - ❑ If a centralized register file were used, the units would have to read their results from the registers when register buses are available
- 2. Elimination of stalls for WAW and WAR hazards**

Tomasulo Drawbacks

- Complexity
- Many associative stores (CDB) at high speed
- Performance limited by Common Data Bus
 - Each CDB must go to multiple functional units
⇒ high capacitance, high wiring density
 - Number of functional units that can complete per cycle limited to one!
 - Multiple CDBs ⇒ more FU logic for parallel assoc stores
- Non-precise interrupts!
 - We will address this later

Final notes

- Next time:
 - More instruction scheduling issues
 - Speculation
 - Multiple issue
 - Multithreading
 - Midterm preview
 - Midterm exam in class Thursday, 3/13
- Announcements/reminders
 - HW 3 due today
 - HW 4 to be posted; due 3/6