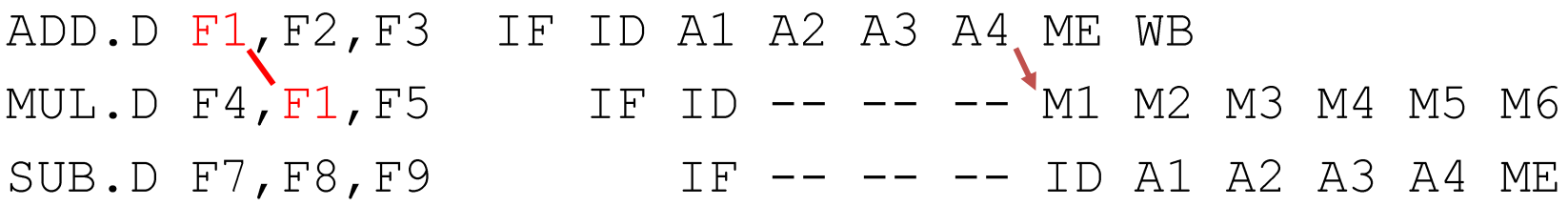
Tutorial 6

Dynamic Scheduling

Tomasulo Algorithm

Quick overview of dynamic scheduling

- Recall that whenever an instruction stalls on the MIPS 5-stage pipeline, *all* instructions after it must also stall.



- SUB.D isn't dependent on either ADD.D or MUL.D. Having to stall SUB.D here is inefficient.
- An instruction should not need to stall if its operands are already available.

Dynamic scheduling (cont'd)

- *Dynamic scheduling* allows independent instructions behind a stall to proceed. If we added hardware for dynamic scheduling to our pipeline, we could have:

```
ADD.D F1,F2,F3   IF ID A1 A2 A3 A4 ME WB
MUL.D F4,F1,F5      IF ID -- -- -- M1 M2 M3 M4 M5 M6
SUB.D F7,F8,F9         IF ID A1 A2 A3 A4 ME WB
```

- With dynamic scheduling, we allow instructions to *execute and complete out of order*.
- An instruction can begin execution as soon as its operands are available.
- ***Tomasulo's algorithm*** implements dynamic scheduling

Tomasulo's algorithm

- Now we **depart from the simple "IF ID EX MEM WB" 5-stage pipeline**
- Each instruction in Tomasulo's algorithm has *3 stages*:
  1. ***Issue***: send instruction to an appropriate "reservation station". If a reservation station is not available, then we must stall this instruction (and all instructions behind it – so issuing of instructions is done in order). After being issued, the instruction is referred to by the name of the reservation station it occupies.
  2. ***Execute***: if operands are available, then start execution; otherwise, wait for the operands to become available and then start execution. Therefore, the execution of instructions may be out of order.
  3. ***Write result***: send execution results to a common data bus. Any instructions (and registers) waiting for this result will pick it up from the bus. The Write Result stage may also be *out of order*.

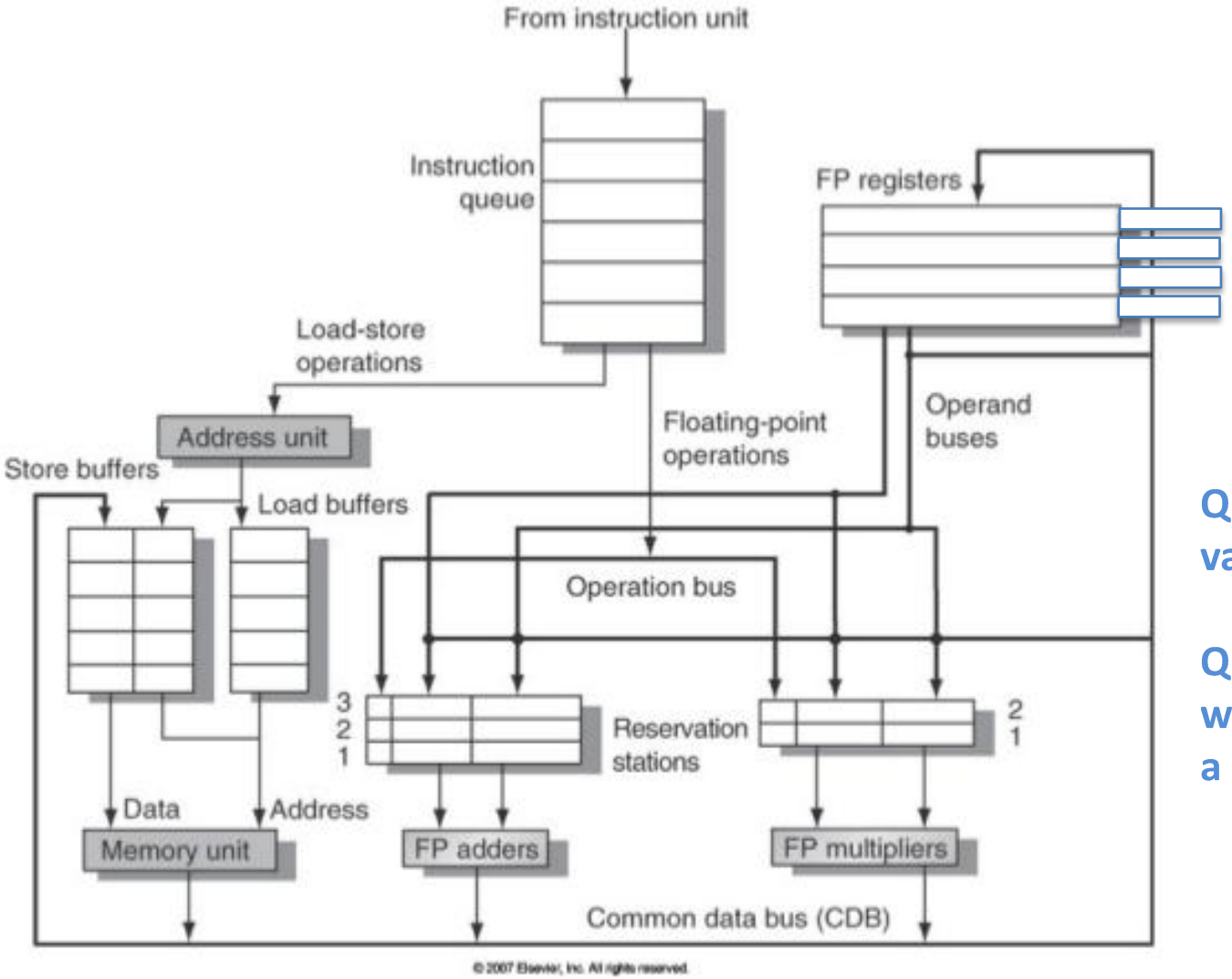
More on Execute stage

What happens in the execute stage for different instruction types ?

- ***ALU/logical instructions***: perform the ALU/logical operation, such as add, divide, etc.
- ***Loads***: calculate address and then load data from memory.
- ***Stores***: calculate address and then store data to memory.

Both loads and stores require an additional step to compute the effective address.

Tomasulo Hardware



RegisterStatus: Qi fields

Qi = 0 means actual value in register file

Qi != 0 means result will be produced by a reservation station

Loop Example

Instruction status:

ITER	Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
1	LD	F0	0	R1			
1	MULTD	F4	F0	F2			
1	SD	F4	0	R1			

	Busy	*Addr*	*Fu*
Load1	No		
Load2	No		
Load3	No		
Store1	No		
Store2	No		
Store3	No		

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	**R1**	*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
0	*Fu*									

Reservation Stations

- Several fields per reservation station
  - OP : the operation
  - Qj, Qk: The reservation stations that produce the value used for the operation
  - Vj, Vk: The actual value of the source operand
  - A: offset/effective address for LD/ST
  - Busy: if reservation station is being used

- At any given time, either the V or the Q field is active for each operand. When a Q value becomes available, Q is set to 0 and the value is written in V. When both Vi and Vk are available in the reservation station, the operation can be carried out in the functional unit corresponding to OP.

- The register file also has a field Qi which tells which reservation station will produce the result to be stored, when it gets broadcasted.

Register Result Status

- Works in parallel with the register file
- Contains a Qi value for each register
  - If Qi=0, the register file contains the required value
  - If Qi!=0, the register file is waiting for the result of a computation.
- When issuing an instruction, the algorithm checks the Register Result Status for each argument of the operation. If Qi=0, take the value from the register file and save in Vi. If Qi!=0, copy Qi to the reservation station so it gets the value too when it gets produced.

Functional Units

- Several functional units can perform various types of operations in parallel if the arguments are available. e.g.: Memory unit, FP add, FP multiply.
- Each functional unit can perform its operation using more than one clock cycle (e.g.: add=2, mult=5, div=20, ld/hit=1, ld/miss=7), which can introduce structural hazards.
- Only a single instruction targeting a given functional unit can proceed at any given time.
- Only a single result from any functional unit can be broadcasted through the Common Data Bus (CDB) at any given time, which can introduce structural hazards.
- Loads and stores require two steps: 1- effective address computation, 2- place the operation in a LD or ST buffer

Common Data Bus (CDB)

- Connects the output of the Functional Units to all blocks which are expecting those results.
- Transmitting a result through the CDB uses 1 clock cycle, but every consumer gets the value at once.
- The bus cannot handle more than one write at a time. If several instructions want to write to the bus in the same cycle, the earliest instruction (in program order) will get the bus.
- The implicit Register renaming of Tomasulo removes the possibility of WAR and WAW hazards.
- Since all instructions are issued in program order, true dependencies are preserved.

Load/Store Dependencies

- Loads and stores require an additional clock cycle to compute the effective address (addition of imm + an integer register value) in the Address Unit
- Stores also have Vi/Qi fields, which holds the value to be stored in memory, or the reservation station which will produce it
- Loads and stores can safely be done out of order if they access different addresses.
- If they have the same address:
  - Interchanging a LD, ST sequence => WAR hazard
  - Interchanging a ST, LD sequence => RAW hazard
  - Interchanging a ST, ST sequence => WAW hazard
- (Loads can always be re-ordered.)

Load/Store Dependencies

- Before executing memory accesses out of order, need to know their effective addresses.
- One way: Perform effective address calculation in program order.
- Other way: When the CPU supports speculation, we can choose to assume that addresses are different and verify later.

Problems with Tomasulo

- Imprecise exceptions
- Limited overlapping of adjacent basic blocks
- Solution?
  - Hardware speculation: Speculative Tomasulo
    - New pipeline stage: **Instruction commit**
    - New hardware: Reorder Buffers (ROB)
    - Out-of-order execution, in-order completion