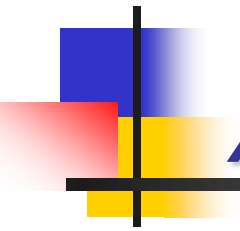
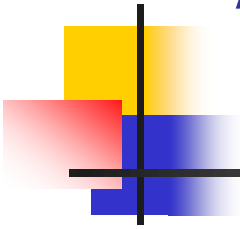


Computer Organization and Architecture

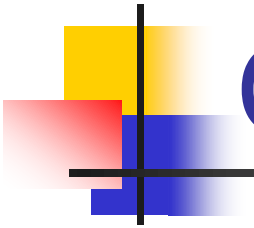
Instructor: Dr. Rushdi Abu Zneit

Slide Sources: Based on CA: aQA by Hennessy/Patterson.

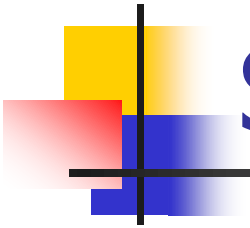


Advanced Topic: Hardware-Based Speculation CA:aQA Sec. 3.7

Speculating on Branch Outcomes



- To optimally exploit ILP (instruction-level parallelism) – e.g. with pipelining, Tomasulo, etc. – it is critical to efficiently maintain *control dependencies* (=branch dependencies)
- Key idea: *Speculate* on the outcome of branches(=predict) and execute instructions *as if* the predictions are correct
  - of course, we must proceed in such a manner as to be able to recover if our speculation turns out wrong
- Three components of hardware-based speculation
  - *dynamic branch prediction* to pick branch outcome
  - *speculation* to allow instructions to execute before control dependencies are resolved, i.e., before branch outcomes become known – with ability to undo in case of incorrect speculation
  - *dynamic scheduling*

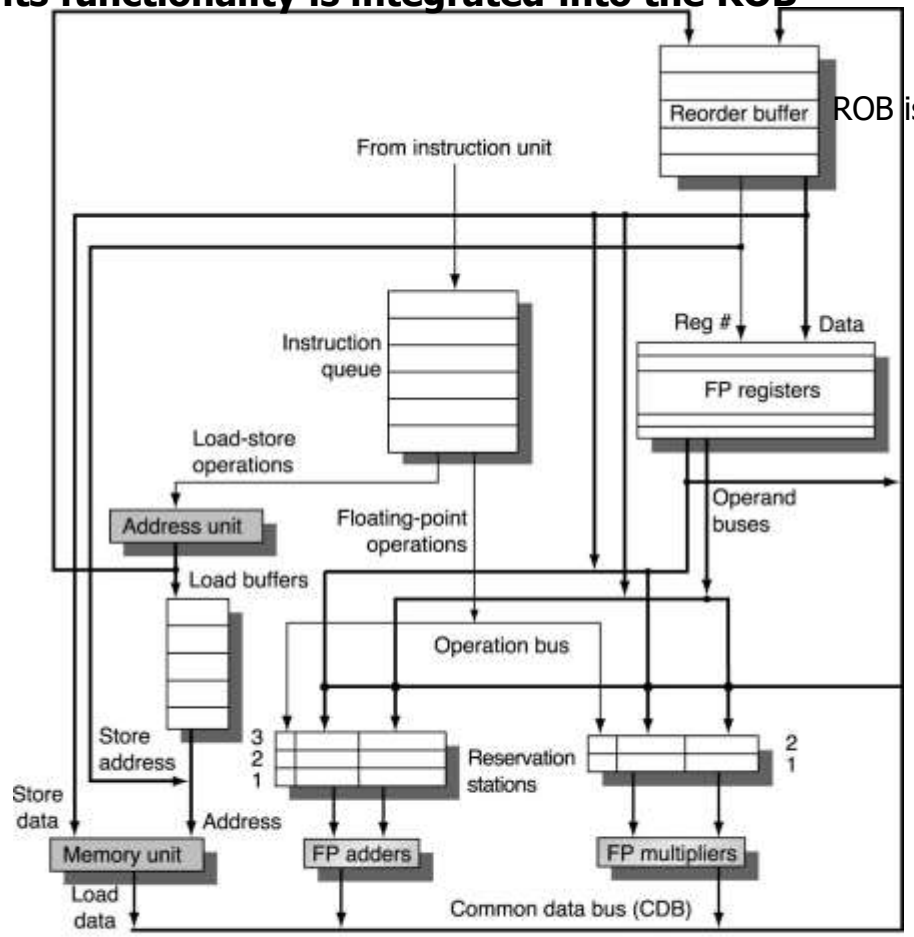


Speculating with Tomasulo

- Modern processors such as PowerPC 603/604, MIPS R10000, Intel Pentium II/III/4, Alpha 21264 *extend Tomasulo's approach to support speculation*
- Key ideas:
  - *separate execution from completion*: allow instructions to execute speculatively but *do not let instructions update registers or memory until they are no longer speculative*
  - therefore, add a final step – after an instruction is no longer speculative – when it is allowed to make register and memory updates, called *instruction commit*
  - *allow instructions to execute and complete out of order but force them to commit in order*
  - add a hardware buffer, called the *reorder buffer* (*ROB*), with registers to hold the result of an instruction *between completion and commit*

Tomasulo Hardware with Speculation

Basic structure of MIPS floating-point unit based on Tomasulo and extended to handle speculation: ROB is added and store buffer in original Tomasulo is eliminated as its functionality is integrated into the ROB



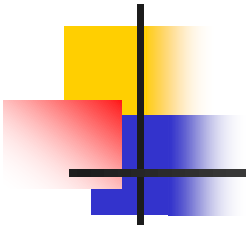
ROB is a queue!

Tomasulo's Algorithm with Speculation: Four Stages

Compare Tomasulo's Algorithm:
Red text = differences!

1. Issue: get instruction from Instruction Queue
   - if reservation station and ROB slot free (no structural hazard), control issues instruction to reservation station and ROB, and sends to reservation station operand values (or reservation station source for values) as well as allocated ROB slot number
2. Execution: operate on operands (EX)
   - when both operands ready then execute; if not ready, watch CDB for result
3. Write result: finish execution (WB)
   - write on CDB to all awaiting units and ROB; mark reservation station available
4. Commit: update register or memory with ROB result
   - when instruction reaches head of ROB and results present, update register with result or store to memory and remove instruction from ROB
   - if an incorrectly predicted branch reaches the head of ROB, flush the ROB, and restart at correct successor of branch

ROB Data Structure



ROB entry fields

- **Instruction type**: branch, store, register operation (i.e., ALU or load)
- **State**: indicates if instruction has completed and value is ready
- **Destination**: where result is to be written – register number for register operation (i.e. ALU or load), memory address for store
  - branch has no destination result
- **Value**: holds the value of instruction result till time to commit

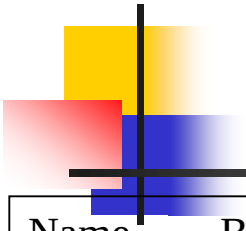
Additional reservation station field

- **Destination**: Corresponding ROB entry number

Example

```
1.   L.D      F6,  34(R2)
2.   L.D      F2,  45(R3)
3.   MUL.D    F0,  F2, F4
4.   SUB.D    F8,  F2, F6
5.   DIV.D    F10, F0, F6
6.   ADD.D    F6,  F8, F2
```

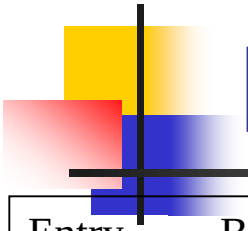
- *Assume latencies load 1 clock, add 2 clocks, multiply 10 clocks, divide 40 clocks*
- *Show data structures just before* `MUL.D` *goes to commit…*



Reservation Stations

Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
Load1	no							
Load2	no							
Add1	no							
Add2	no							
Add3	no							
Mult1	yes	MUL	Mem[45+Regs[R3]]	Regs[F4]			#3	
Mult2	yes	DIV		Mem[34+Regs[R2]]	#3		#5	

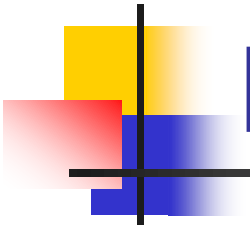
Addi indicates i*th* reservation station for the FP add unit, etc.



Reorder Buffer

Entry	Busy	Instruction		State	Destination	Value
1	no	L.D	F6, 34(R2)	Commit	F6	Mem[34+Regs[R2]]
2	no	L.D	F2, 45(R3)	Commit	F2	Mem[45+Regs[R3]]
3	yes	MUL.D	F0, F2, F4	Write result	F0	#2 × Regs[F4]
4	yes	SUB.D	F8, F6, F2	Write result	F8	#1 – #2
5	yes	DIV.D	F10, F0, F6	Execute	F10	
6	yes	ADD.D	F6, F8, F2	Write result	F6	#4 + #2

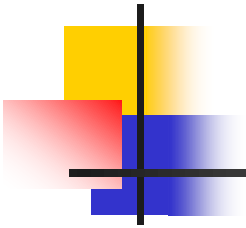
At the time `MUL.D` is ready to commit only the two `L.D` instructions have already committed, though others have completed execution
Actually, the `MUL.D` is at the head of the ROB – the `L.D` instructions are shown only for understanding purposes
#X represents value field of ROB entry number X



Registers

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

Floating point registers



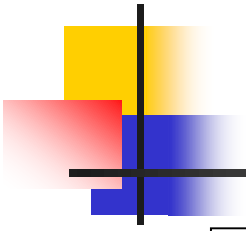
Example

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

- *Assume instructions in the loop have been issued twice*
- *Assume* `L.D` *and* `MUL.D` *from the first iteration have committed and all other instructions have completed*
- *Assume effective address for store is computed prior to its issue*
- *Show data structures…*

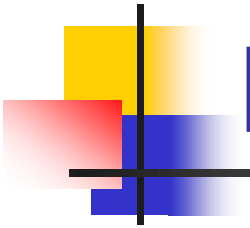
Reorder Buffer

Entry	Busy	Instruction		State	Destination	Value
1	no	L.D	F0, 0(R1)	Commit	F0	Mem[0+Regs[R1]]
2	no	MUL.D	F4, F0, F2	Commit	F4	#1 × Regs[F2]
3	yes	S.D	F4, 0(R1)	Write result	0 + Regs[R1]	#2
4	yes	DADDUI	R1, R1, #-8	Write result	R1	Regs[R1] – 8
5	yes	BNE	R1, R2, Loop	Write result		
6	yes	L.D	F0, 0(R1)	Write result	F0	Mem[#4]
7	yes	MUL.D	F4, F0, F2	Write result	F4	#6 × Regs[F2]
8	yes	S.D	F4, 0(R1)	Write result	0 + #4	#7
9	yes	DADDUI	R1, R1, #-8	Write result	R1	#4 – 8
10	yes	BNE	R1, R2, Loop	Write result		



Registers

Field	F0	F1	F2	F3	F4	F5	F6	F7	F8
Reorder#	6				7				
Busy	yes	no	no	no	yes	no	no	…	no



Notes

- If a branch is mispredicted, recovery is done by flushing the ROB of all entries that appear after the mispredicted branch
  - entries before the branch are allowed to continue
  - restart the fetch at the correct branch successor
- When an instruction commits or is flushed from the ROB then the corresponding slots become available for subsequent instructions