Advanced Computer Architecture
高级计算机体系结构

第3讲： ISA and ILP (3)

张献伟

xianweiz.github.io

DCS5367, 10/12/2021

Review Questions (1)

- Five-stage execution?

  Inst fetch (IF), Inst decode (ID), Execution (EX), Mem access (MEM), Write back (WB)

- Stages of ‘add R3, R1, R2’ ?

  IF, ID, EX, WB

- What is Pipelining?

  Multi instructions are overlapped in execution

- Ideal speedup of pipelining?

  N (number of stages)

- Impossible to reach the ideal speedup, why?

  Imbalanced stages, pipelining overhead

- Pipeline hazards?

  Structural, data, control

Review Questions (2)

- Explain data hazard.
  Pipeline changes the order of read/write accesses to operands
- How to avoid data hazards?
  Forwarding
- Is forwarding sufficient to clear all data hazards?
  Nope. Stalls may be needed.
- Cause of branch hazard?
  Branch has a delay in determining the proper inst to fetch
- Types of dependences.
  Data dependence, name dependence (anti & output)
- How to remove name dependences?
  Register renaming

Control Dependences[控制依赖]

- Determine the order of instructions with respect to branches[相对分支的指令顺序]

if *P1* then *S1* ;	*S1* is control dependent on *P1* and
if *P2* then *S2* ;	*S2* is control dependent on *P2* (and P1 ??)

- An instruction that is control dependent on *P* cannot be moved to a place where it is no longer control dependent on *P*, and visa-versa[不可移动]

```
Example 1:
    add    x1, x2, x3
    beq    x4, x0, L
    sub    x1, x5, x6
L: …
    or     x7, x1, x8
```

"or" depends on the execution flow

```
Example 2:
    add    x1, x2, x3
    beq    x12, x0, skip
    sub    x4, x5, x6
    add    x5, x4, x9
skip:
    or     x7, x8, x9
```

possible to move "sub" before "beq" (if x4 is not used after skip)

Compiler Techniques to Expose ILP

- Scheduling[调度]
  - To keep a pipeline full, parallelism among insts must be exploited by finding sequences of unrelated insts that can be overlapped in the pipeline[重叠]
  - To avoid a pipeline stall, the execution of a dependent inst must be separated from the source insts by a distance in clock cycles equal to the pipeline latency of that source inst[分隔]
- A compiler’s ability to perform the scheduling depends on
  - Amount of ILP in the program[程序特性]
  - Latencies of the functional units in the pipeline[硬件特性]
- Compiler can increase the amount of available of ILP by transforming loops[循环转换]

Loop Dependences(§3.2) [循环依赖]

```
for (i = 999; i >= 0; i = i-1)
    x[i+1] = x[i] + y[i];
```

- [有]There is a loop carried dependence since the statement in an iteration depends on an earlier iteration

```
for (i = 999; i >= 0; i = i-1)
    x[i] = x[i] + s;
```

- [无]There is no loop carried dependence

- The iterations of a loop can be executed in parallel if there is no loop carried dependence

中山大學 SUN YAT-SEN UNIVERSITY

Example: Loop Transformation[循环转换]

```
for (i = 999; i >= 0; i = i-1)
    x[i] = x[i] + s;
```

```
Loop:  fld      f0, 0(x1)       //f0=array element
       fadd.d   f4, f0, f2      //add scalar in f2
       fsd      f4, 0(x1)       //store result
       addi     x1, x1, -8      //decrement pointer
                                //8 bytes (per DW)
       bne      x1, x2, Loop    //branch x1 != x2
```

- Assume the latencies of FP operations
  - 3 cycles if an **FP ALU op** follows and depends on an **FP ALU op**
  - 2 cycles if an **FP store** follows and depends on an **FP ALU op**
  - 1 cycle is an **FP ALU op** follows and depends on an **FP load**
  - 1 cycle if a **branch** follows and depends on on **Integer ALU op**

Basic Scheduling[简单调度]

- Re-order the statements
  - Actual work: *load*, *add* and *store*
  - loop overhead: *addi*, *bne*, two *stalls*

			cycle
Loop:	fld	f0, 0(x1)	1
	stall		2
	fadd.d	f4, f0, f2	3
	stall		4
	stall		5
	fsd	f4, 0(x1)	6
	addi	x1, x1, -8	7
	stall		8
	bne	x1, x2, loop	9

9 clock cycles per iteration

			cycle
Loop:	fld	f0, 0(x1)	1
	addi	x1, x1, -8	2
	fadd.d	f4, f0, f2	3
	stall		4
	stall		5
	fsd	f4, 8(x1)	6
	bne	x1, x2, loop	7

7 clock cycles per iteration

Loop Unrolling[循环展开]

- Simply replicates the loop body multiple times, adjusting the loop termination code[复制->调整]
  - Increases the number of insts relative to the branch and overhead insts[增加有效指令数]
  - Eliminates branches, thus allowing insts from different iterations to be scheduled together[消除分支，共同调度]

```
Loop:  fld     f0, 0(x1)
       fadd.d  f4, f0, f2
       fsd     f4, 0(x1)
       addi    x1, x1, -8
       bne     x1, x2, loop
```

```
Loop:  fld     f0, 0(x1)
       fadd.d  f4, f0, f2
       fsd     f4, 0(x1)          //drop addi & bne
       fld     f6, -8(x1)
       fadd.d  f8, f6, f2
       fsd     f8, -8(x1)         //drop addi & bne
       fld     f0, -16(x1)
       fadd.d  f12, f0, f2
       fsd     f12, -16(x1)       //drop addi & bne
       fld     f14, -24(x1)
       fadd.d  f16, f14, f2
       fsd     f16, -24(x1)       //drop addi & bne
       addi    x1, x1, -32
       bne     x1, x2, loop
```

Loop Unrolling[循环展开]

- Simply replicates the loop body multiple times, adjusting the loop termination code[复制->调整]
  - Increases the number of insts relative to the branch and overhead insts[增加有效指令数]
  - Eliminates branches, thus allowing insts from different iterations to be scheduled together[消除分支，共同调度]

```
Loop:  fld     f0, 0(x1)
       fadd.d  f4, f0, f2
       fsd     f4, 0(x1)
       fld     f6, -8(x1)
       fadd.d  f8, f6, f2
       fsd     f8, -8(x1)
       fld     f0, -16(x1)
       fadd.d  f12, f0, f2
       fsd     f12, -16(x1)
       fld     f14, -24(x1)
       fadd.d  f16, f14, f2
       fsd     f16, -24(x1)
       addi    x1, x1, -32
       bne     x1, x2, loop
```

```
Loop:  fld     f0, 0(x1)
       fld     f6, -8(x1)
       fld     f0, -16(x1)
       fld     f14, -24(x1)
       fadd.d  f4, f0, f2
       fadd.d  f8, f6, f2
       fadd.d  f12, f0, f2
       fadd.d  f16, f14, f2
       fsd     f4, 0(x1)
       fsd     f8, -8(x1)
       fsd     f12, -16(x1)
       fsd     f16, -24(x1)
       addi    x1, x1, -32
       bne     x1, x2, loop
```

A total of 14 clock cycles (3.5 cycles per element)

Unrolling Limitations[限制]

- The gains from loop unrolling are limited by
  - A decrease in the amount of overhead amortized with each unroll
    - Unrolled 4 times → 8 times: ½ cycle/element → ¼ cycle/element
  - Growth in code size caused by unrolling
    - May increase in the inst cache miss rate
    - May bring register pressure (more live values)
  - Compiler limitations
    - Sophisticated transformations increases the compiler complexity

```
Loop:  fld     f0, 0(x1)
       fld     f6, -8(x1)
       fld     f0, -16(x1)
       fld     f14, -24(x1)
       fadd.d  f4, f0, f2
       fadd.d  f8, f6, f2
       fadd.d  f12, f0, f2
       fadd.d  f16, f14, f2
       fsd     f4, 0(x1)
       fsd     f8, -8(x1)
       fsd     f12, -16(x1)
       fsd     f16, -24(x1)
       addi    x1, x1, -32
       bne     x1, x2, loop
```

Branch Prediction(§3.3)[分支预测]

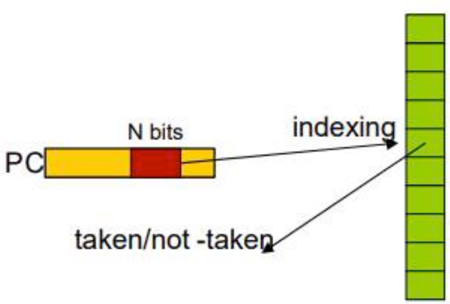
- Branches hurt pipeline performance
  - Branch hazards and stalls
- Static branch prediction[静态分支预测]
  - The default is to assume that branches are not taken
  - May have a design which predicts that branches are taken
- Reasonable to assume that[假设]
  - Forward branches are often not taken
  - Backward branches are often taken
- More predictors based on branch directions
  - Profiling is the standard technique for predicting the probability of branching
  - Dynamic predictors rely on the history to predict the future branch direction

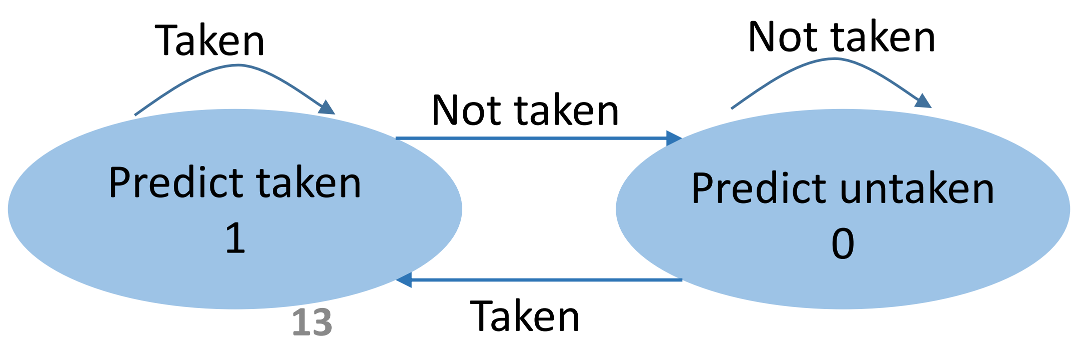
```
    add     x1, x2, x3
    beq     x4, x0, L
    sub     x1, x5, x6
L: ...
    or x7, x1, x8
```

```
    add   x1, x2, x3
skip:
    or    x7, x8, x9
    beq   x12, x0, skip
    sub   x4, x5, x6
```

Dynamic Branch Prediction(§C2.7)[动态]

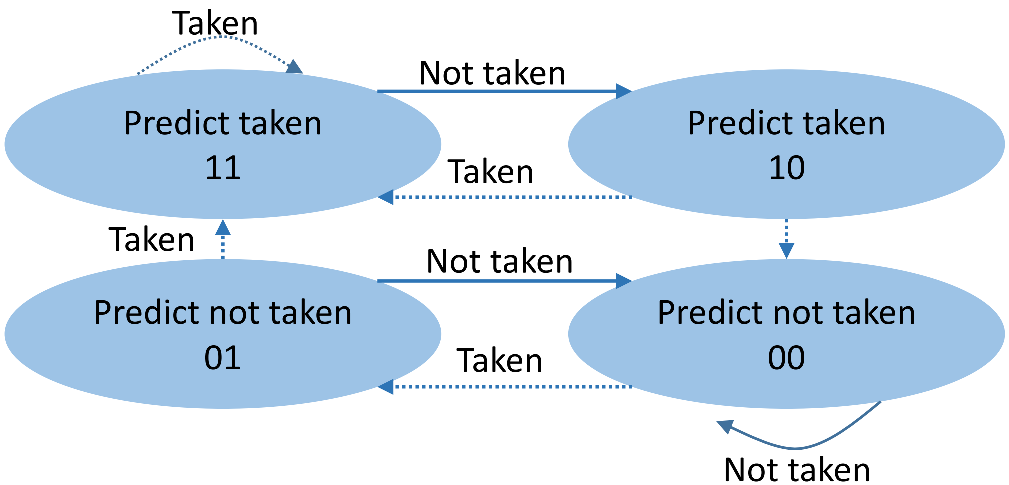
- Performance depends on the accuracy of prediction and the cost of miss-prediction[性能影响]
- The simplest branch prediction scheme: **Branch Prediction Buffer**[分支预测缓存]:
  - 1-bit table (cache) indexed by some bits of the address of the branch instructions (can be accessed in decode stage) -> hashing[指令地址的低位作为索引]
  - Record whether or not the branch was taken last time – may have collision[冲突]
  - Will cause two miss-predictions in a loop (at start and end of loop)





Two-bit Branch Predictors

- Change your prediction only if you miss-predict twice[稳定性]
  - A branch that strongly favors take or not taken (many branches do), will be miss-predicted less often than with a 1-bit predictor



- In general, *n*-bit predictors are called **Local Predictors**[局部预测器]
  - Use a saturated counter (++ on correct prediction, -- on wrong prediction)
  - *n*-bit prediction is not much better than 2-bit prediction (n > 2).
  - A BHT with 4K entries is as good as an infinite size BHT[无限缓冲区]

Correlating Branch Predictors[关联预测]

- Hypothesis[假设]: recent branches are correlated (behavior of recently executed branches affects prediction of current branch)
- Example 1:

```
if (aa==2)
      aa=0;
if (bb==2)
      bb=0;
if (aa!=bb) {
```

```
      addi  x3,x1, -2
      bnez  x3, L1 ...      //B1 (aa != 2)
      add   x1, x0, x0      //aa=0
L1:   addi  x3, x2, -2
      bnez  x3, L2          //B2 (bb != 2)
      add   x2, x0, x0      //bb=0
L2:   sub   x3, x1, x2      //x3=aa-bb
      beqz  x3, L3          //B3 (aa == bb)
```

If B1 is not taken (aa==2) and B2 is not taken (bb==2), then B3 will be taken (aa==bb)

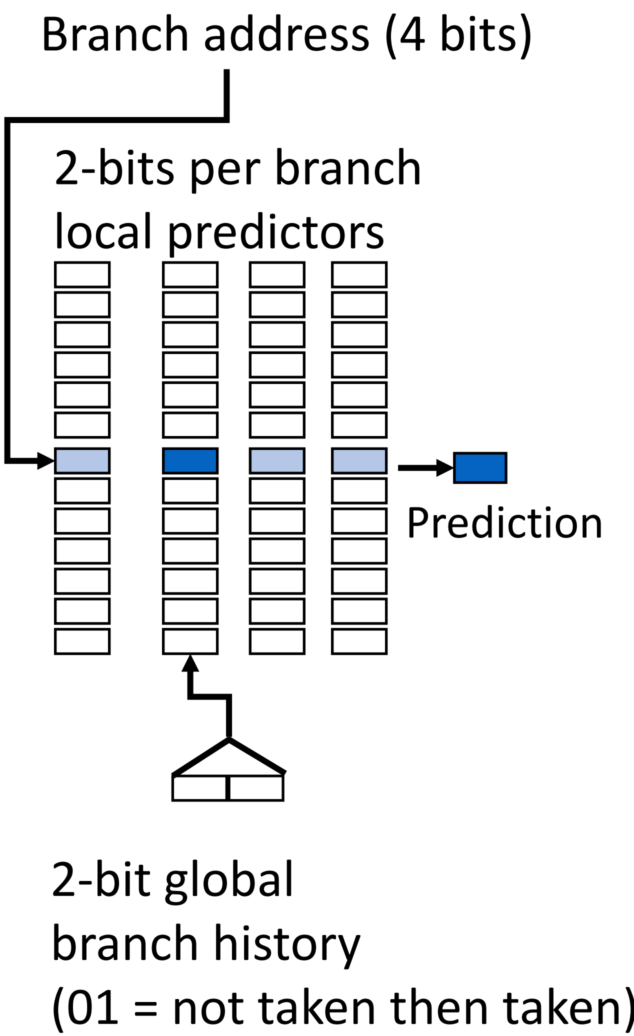
If B1 and B2 are taken (aa!=2, bb!=2), then B3 will probably not be taken

- Example 2:

```
if (d == 0)  d = 1 ;
if (d == 1)  .....
```

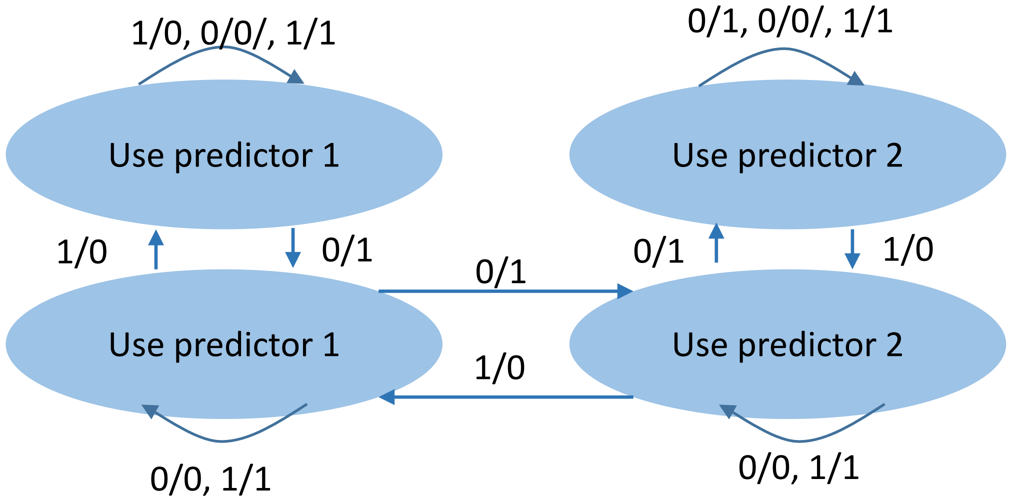
Correlating Branch Predictors (cont.)

- Keep history of the *m* most recently executed branches in an *m*-bit shift register[移位寄存器]
  - Record the prediction for each branch inst, and each of the 2^m combinations
- In general, *(m,n)* predictor means record last *m* branches to select between 2^m history tables each with *n*-bit predictor
  - Simple access scheme (double indexing).
  - A *(0,n)* predictor is a local *n*-bit predictor.
- Size of table is $N*n*2^m$
  - *N* is the number of table entries
  - There is a tradeoff between *N* (determines collision), *n* (accuracy of local prediction) and *m* (determines history)



Tournament predictor[竞赛预测器]

- Combines a global predictor and a local predictor with a strategy for selecting the appropriate predictor (*multi-level* predictors)

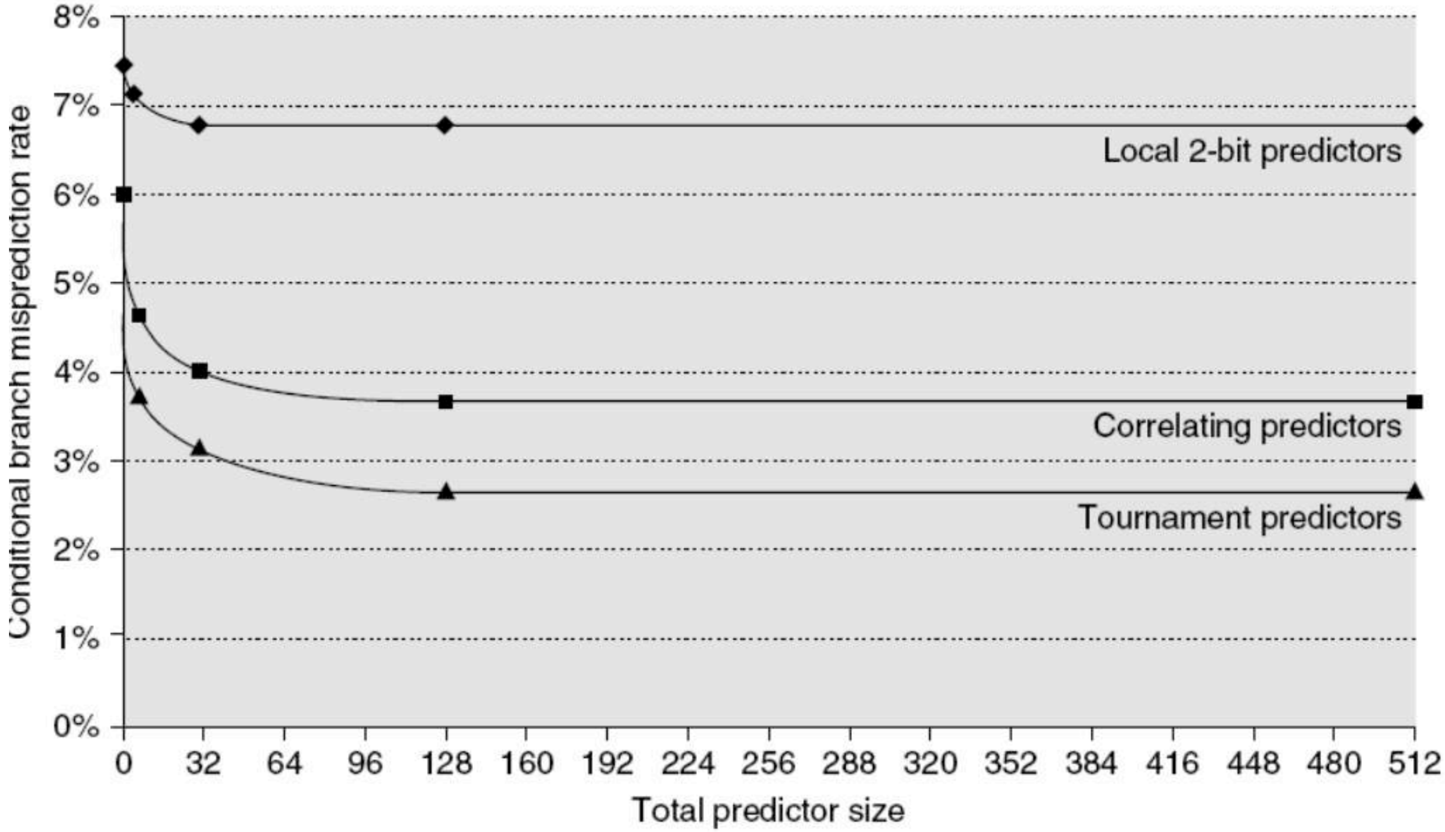


p1/p2 == predictor 1 is correct/ predictor 2 is correct

- The Alpha 21264 selects between
  - A (12,2) global predictor with 4K entries
  - A local predictor which selects a prediction based on the outcome of the last 10 executions of any given branch.

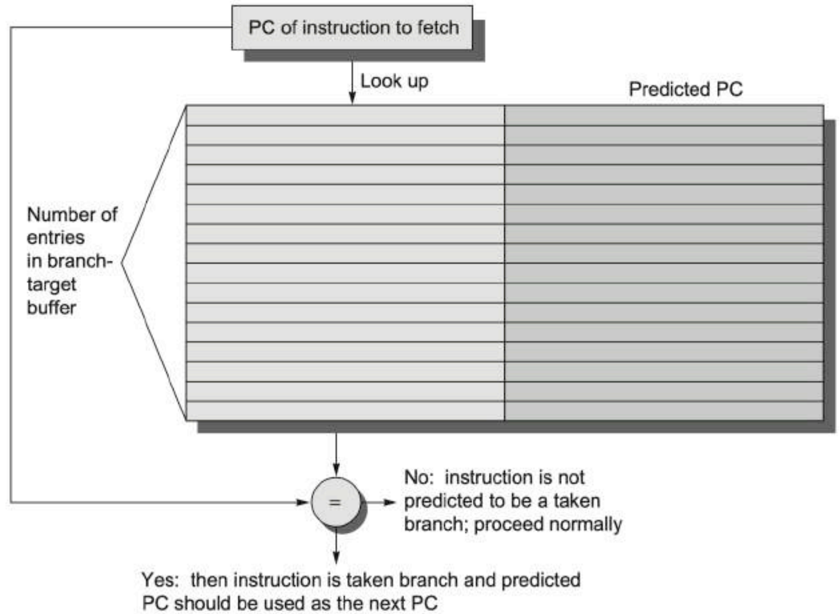
Performance[性能]

- Miss prediction rate for three different predictors



Branch Target Buffers(§3.9)[目标缓冲区]

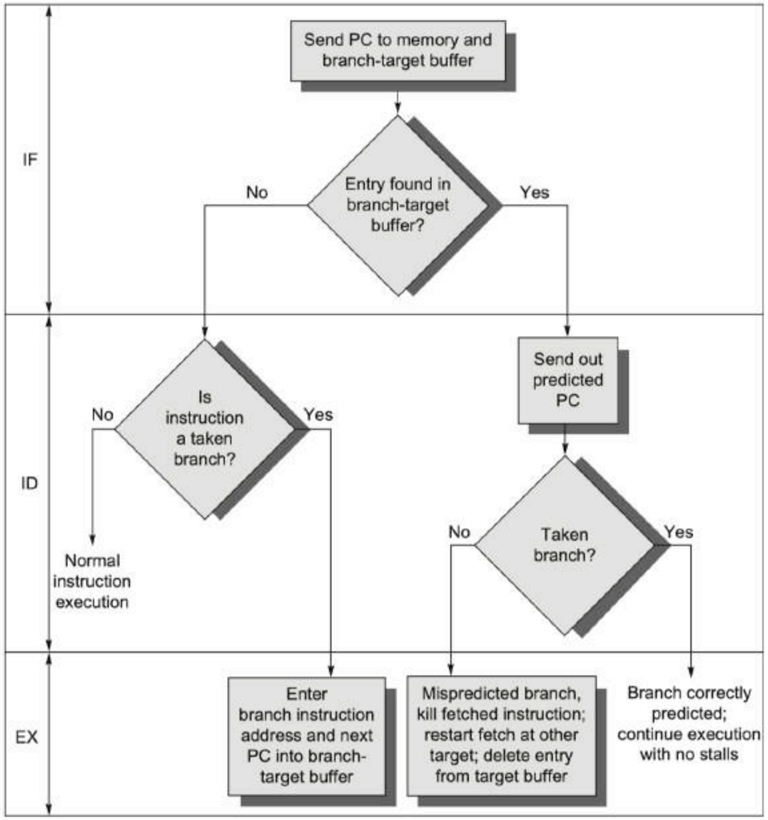
- To increase instruction fetch bandwidth
  - Store the *address* of the branch's target, in addition to the prediction



- Can determine the target address while fetching the branch instruction
  - How do you even know that the instruction is a branch?
  - Can't afford to use wrong branch address due to collision -- why?

Branch Prediction & Pipelining

- Assuming that branch condition and target are resolved in *ID* stage



- A similar chart may be drawn if branch condition/target are resolved in *EX*

Evaluation example

- Assume
  - access branch target buffer in IF stage
  - branch condition determined in ID
  - branch address determined in EX stage
- What is the branch penalty if:
  - penalty for correct prediction = 0 cycle
  - penalty for wrong prediction = 1 (or 2) cycles for non-taken (or taken) branch (assuming that target is not stored in BTB if “predict not taken”)
  - penalty if cannot predict and the branch is taken = 2 cycles
  - branch taken frequency = 60%
  - BTB hit rate = 80% (assume not taken in case of inability to predict)
  - BTB prediction accuracy = 90%
  - The correct instruction is fetched $0.8*0.9+0.2*0.6 = 84\%$ of the time
- May store the target instruction and not only the address - useful when access of table needs more than one cycle.

Dynamically scheduled pipelines (§3.4)

- Key idea: allow instructions behind stall to proceed

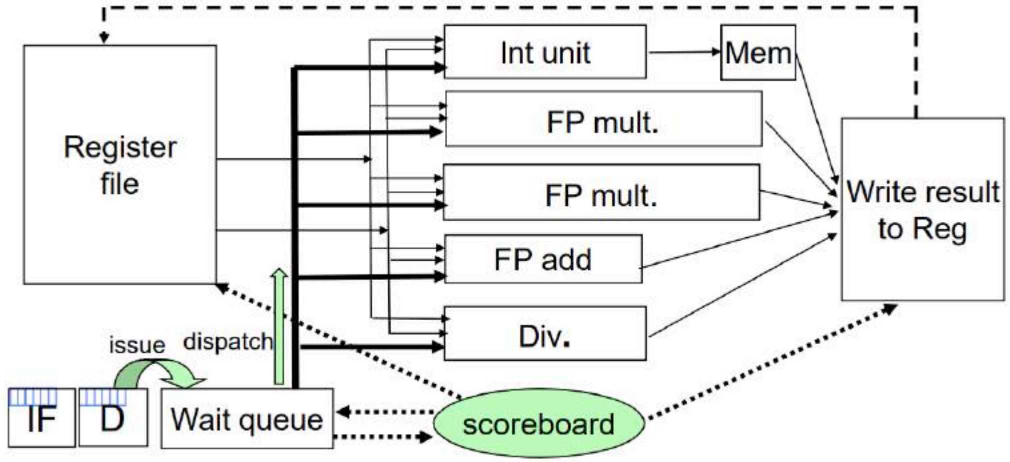
```
fdiv   F0,   F2,   F4
fadd   F10,  F0,  F8
fsub   F12,  F8,   F14
```

RAW -> Stall

No dependency

  - Enables out-of-order execution,
  - Can lead to out-of-order completion.
- Using Scoreboards[记分板] (§ C.7):
  - Dates to the first supercomputer, the CDC 6600 in 1963
  - Split the ID stage into
    - Issue - decode and check for structural hazards,
    - Read operands - wait until no data hazards, then read operands.
  - Instructions wait in a queue and may move to the EX stage (dispatched) out of order.

A scoreboard architecture



- The scoreboard is responsible for instruction issue and execution, including hazard detection. It is also controlling the writing of the results.
- The "scoreboard" consists of 3 tables to keep track of execution progress and the associated intelligence to determine when to dispatch instructions .
- One entry (buffer) in the "wait queue" is associated with each functional unit.

Scoreboard information (3 tables)

- Instruction status[指令状态]
  - issued, read operands and started execution (dispatched), completed execution or wrote result,
- Functional unit status (assuming non-pipelined units) [功能单元状态]
  - busy/not busy
  - operation (if unit can perform more than one operation)
  - destination register - F_i
  - source registers (containing source operands) - F_j and F_k
  - the unit producing the source operands (if stall to avoid RAW hazards) - Q_j and Q_k
  - flags to indicate that source operands are ready - R_j and R_k
- Register result status[寄存器结果状态]
  - indicates the functional unit that contains an instruction which will write into each register (if any)

Four stages of scoreboard control

- Issue only if no structural, WAR or WAW hazards.
  - Issue (and reserve the functional unit) if the functional unit is free and
    - No issued or dispatched instruction (in state “issued” or “dispatched”) will write to the destination register (to avoid WAW)
    - No issued instruction (in state “issued”) will read from the destination register (to avoid WAR)
  - otherwise, stall, and block subsequent instructions
  - the fetch unit stalls when the queue between the fetch and the issue stages is full (may be only one buffer).
- Read operands only if no RAW hazard.
  - If a RAW hazard is detected, wait until the operands are ready,
  - When the operands are ready, read the registers and move to the execution stage,
  - Note that instructions may proceed to the EX stage out-of-order.
- Execution.
  - When execution terminates, notify the score board.
- Write result to register file

Scoreboard example

	Instruction		Issue	Read op.	Exec. Completed	Write result	
	fld	F6, 34(R2)	X	X	X	X	done
	fld	F2, 45(R3)	X	X	X		
RAW	fmul.d	F0, F2, F4	X				
RAW	fsub,d	F8, F6, F2	X				
RAW	fdiv.d	F10, F0, F12	X				
Structure hazard, WAR	fadd.d	F6, F8, F2					Not in

Instruction status

Func. unit status

Unit	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	Yes	Load	F2	R3				Yes	
Mult1	Yes	Mult	F0	F2	F4	Int.		No	Yes
Mult2	No								
Add	Yes	Sub	F8	F6	F2		Int.	Yes	No
divide	Yes	Div	F10	F0	F12	Mult1		No	Yes

Register status

	F0	F2	F4	F6	F8	F10	F12	…	F30
Func. U	Mult1	Int.			Add	Div			

Scoreboard example

- when “fld F2, 45(R3)” is writing

Instruction status

Instruction		Issue	Read op.	Exec. Completed	Write result	
fld	F6, 34(R2)	X	X	X	X	done
fld	F2, 45(R3)	X	X	X	X	
fmul.d	F0, F2, F4	X				
fsub,d	F8, F6, F2	X				
fdiv.d	F10, F0, F12	X				
fadd.d	F6, F8, F2					Not in

Func. unit status

Unit	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	Yes	Load	F2	R3				Yes	
Mult1	Yes	Mult	F0	F2	F4			Yes	Yes
Mult2	No								
Add	Yes	Sub	F8	F6	F2			Yes	Yes
divide	Yes	Div	F10	F0	F12	Mult1		No	Yes

Register status

	F0	F2	F4	F6	F8	F10	F12	…	F30
Func. U	Mult1				Add	Div			

Scoreboard example

- 3 cycles after “fsub.d” finished writing

Instruction status

Instruction		Issue	Read op.	Exec. Completed	Write result
fld	F6, 34(R2)	X	X	X	X
fld	F2, 45(R3)	X	X	X	X
fmul.d	F0, F2, F4	X	X	X	
fsub,d	F8, F6, F2	X	X	X	X
fdiv.d	F10, F0, F12	X			
fadd.d	F6, F8, F2	X	X	X	

Func. unit status

Unit	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	No								
Mult1	Yes	Mult	F0	F2	F4			Yes	Yes
Mult2	No								
Add	Yes	add	F4	F8	F2			Yes	Yes
divide	Yes	Div	F10	F0	F12	Mult1		No	Yes

Register status

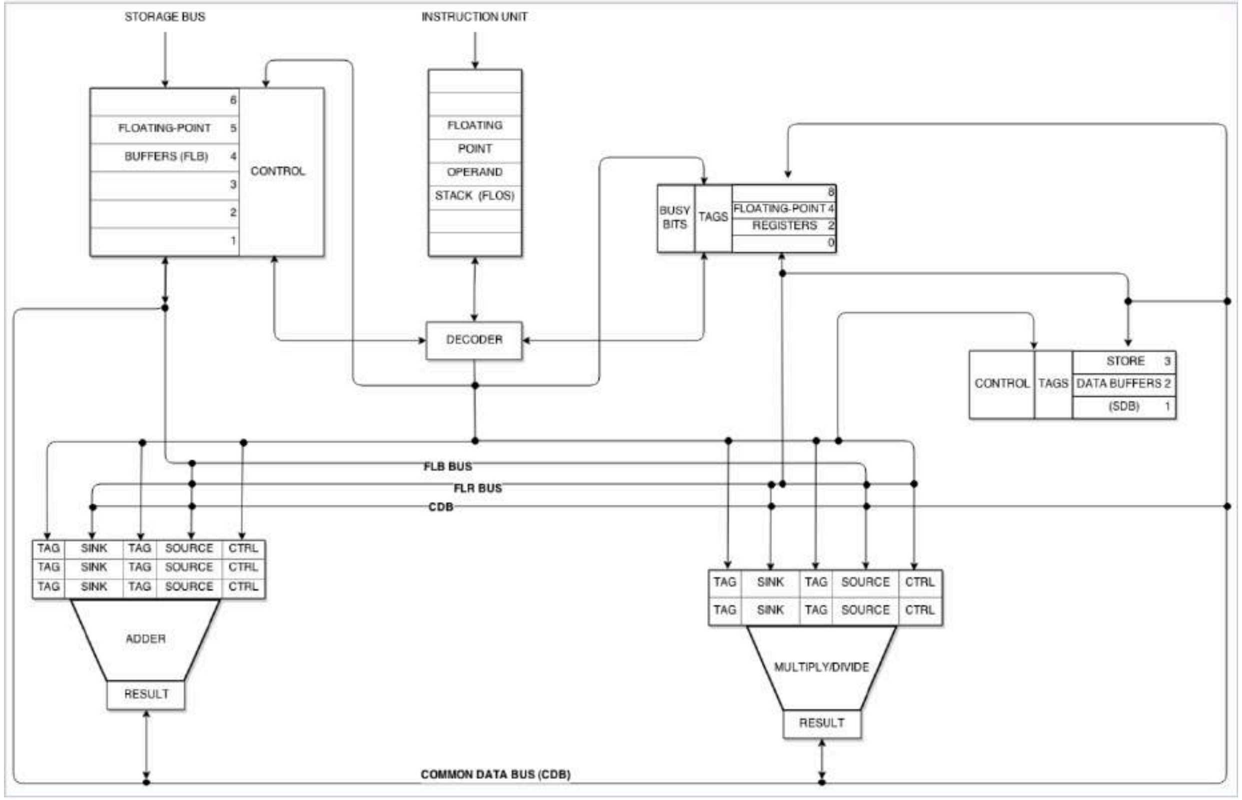
	F0	F2	F4	F6	F8	F10	F12	…	F30
FU	Mult1		Add			Div			

Scoreboard's costs and benefits

- Limitations of the scoreboard approach
  - No forwardingstructural hazards are cleared before instruction "issue"
  - WAW and WAR hazards are cleared before instruction "issue"
  - Did not discuss control hazards
  - Execution units are not pipelined
- Possible enhancement
  - If we can have "k" write-backs to registers per cycle and "k" parallel buses between registers and pipeline units, them
    - k functional units may be released per cycle
    - k instructions may be dispatched per cycles.
    - k instructions may be issued per cycle.
- Need to extend the scoreboard to the case where the execution units are pipelined?

Tomasulo’s algorithm

- A computer arch. Hardware algo. For dynamic scheduling of instructions, allowing OoO. execution. Including:
  - Common Data Bus
  - Instruction Order
  - Register renaming
  - Exceptions



Tomasulo's algorithm

- Step 1:
  - Issue: Instructions are issued for execution if all operands and reservation stations are ready or else they are stalled. Register are renamed in this step, eliminating WAR and WAW hazards.

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			

Tomasulo's algorithm

- Step 2:
  - Execute: The instruction operations are carried out. Instructions are delayed here until all of their operands are available, eliminating RAW hazards.

Instruction	Issue	Execute	Write result
	Instruction status		
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	√		

Name	Busy	Op	Vj	Vk	Qj	Qk	A
	Reservation stations						
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		

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

中山大學 SUN YAT-SEN UNIVERSITY

NSCC GZ

Tomasulo’s algorithm

- Step 3:
  - Write Result: ALU operations results are written back to registers and store operations are written back to memory
    - If the instruction was an ALU operation
      - If the result is available, write it on the CDB and from there into the registers and any reservation stations waiting for this result
    - Else write the data to memory during this step

Tomasulo’s algorithm

- Example:

```
Loop:   fld f0,0(x1)
        fmul.d f4,f0,f2
        fsd f4,0(x1)
        addi x1,x1,8
        bne x1,x2,Loop // branches if x16 != x2
```

Tomasulo's algorithm

- Example:

Instruction	From iteration	Issue	Execute	Write result
	Instruction status			
fld f0,0(x1)	1	√	√	
fmul.d f4,f0,f2	1	√		
fsd f4,0(x1)	1	√		
fld f0,0(x1)	2	√	√	
fmul.d f4,f0,f2	2	√		
fsd f4,0(x1)	2	√		

Name	Busy	Op	Vj	Vk	Qj	Qk	A
	Reservation stations						
Load1	Yes	Load					Regs[x1] + 0
Load2	Yes	Load					Regs[x1] – 8
Add1	No						
Add2	No						
Add3	No						
Mult1	Yes	MUL		Regs[f2]	Load1		
Mult2	Yes	MUL		Regs[f2]	Load2		
Store1	Yes	Store	Regs[x1]			Mult1	
Store2	Yes	Store	Regs[x1] – 8			Mult2	

Field	f0	f2	f4	f6	f8	f10	f12	...	f30
	Register status								
Qi	Load2		Mult2						

Hardware-Based Speculation

- Basic Concept:
  - Overcome control dependence by hardware speculating on outcome of branches and executing program as if guesses were correct
  - If prediction is wrong, it need a hardware to handle it
  - Extension over branch prediction with dynamic scheduling
    - Speculation ⇒ fetch, issue, and execute instructions as if branch predictions were always correct
    - Dynamic scheduling ⇒ only fetches and issues such instructions
  - A data flow execution model: Operations execute as soon as their operands are available

Hardware-Based Speculation

- 3 components
  - Dynamic branch prediction
  - Speculation
  - Dynamic scheduling

- 3 rules
  - Extending Tomasulo's algorithm
  - OoO. execution but in-order commit
  - The register file is not updated until instruction commits

Hardware-Based Speculation

- Key idea
  - Allow instructions to execute OoO.
  - Force instructions to commit in order
  - Prevent any irrevocable action (such as updating state or taking an exception) until an instruction commits

- Hence:
  - Must separate execution from allowing instruction to finish or “commit”
    - Instructions may finish execution considerably before they are ready to commit

Reorder Buffer

- Function:
  - Holds the result of instruction between completion and commit

- Four fields:
  - Instruction type: branch / store / register
  - Destination field: register number
  - Value field: output value
  - Ready field: completed execution

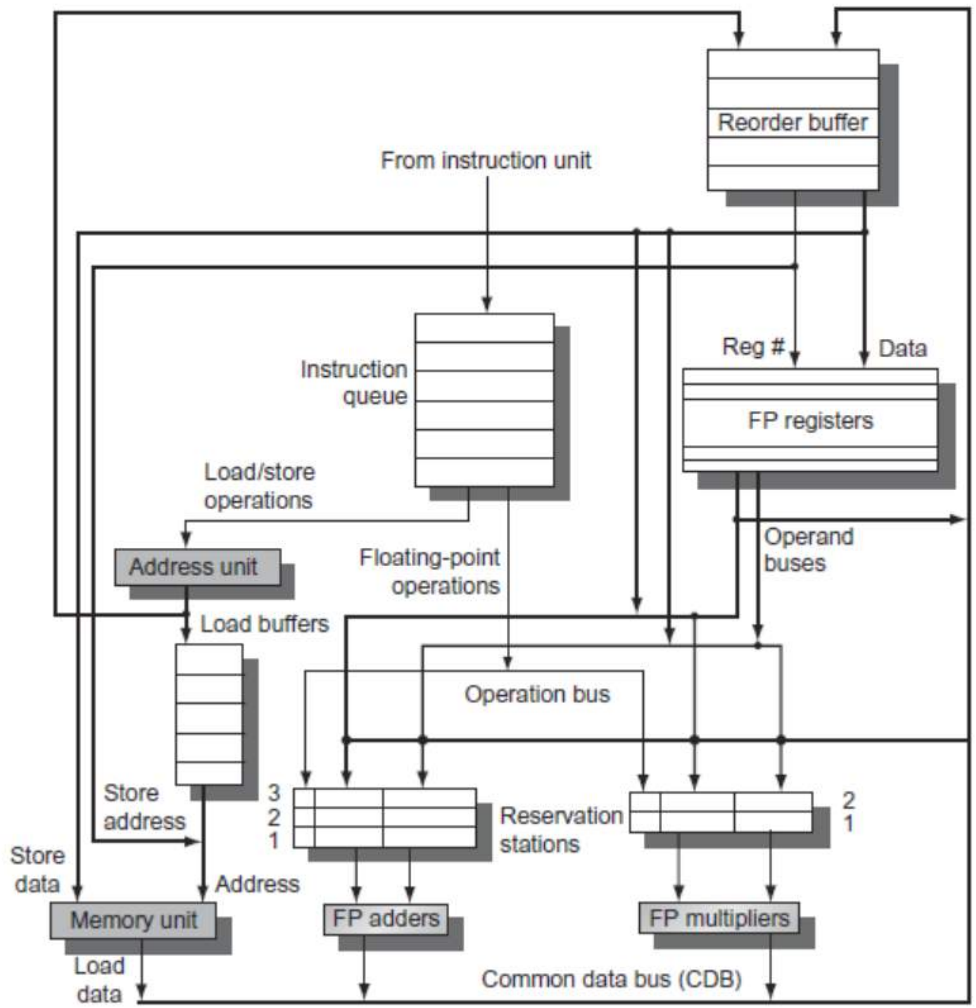
- Modify reservation stations:
  - Operand source is now reorder buffer instead of functional unit

Reorder Buffer Procedure

- Issue:
  - Allocate reservation station(R.S.) and Reorder Buffer(R.O.B), read available operands
- Execute:
  - Begin execution when operand values are available
- Write Result:
  - Write result and R.O.B. tag on C.D.B.
- Commit:
  - When R.O.B. reaches head of R.O.B., update register
  - When a mispredicted branch reaches head of R.O.B., discard all entries

Reorder Buffer

- Register values and memory values are not written until an instruction commits
- On misprediction:
  - Speculated entries in R.O.B. are cleared
- Exceptions:
  - Not recognized until it is ready to commit



Reorder Buffer

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

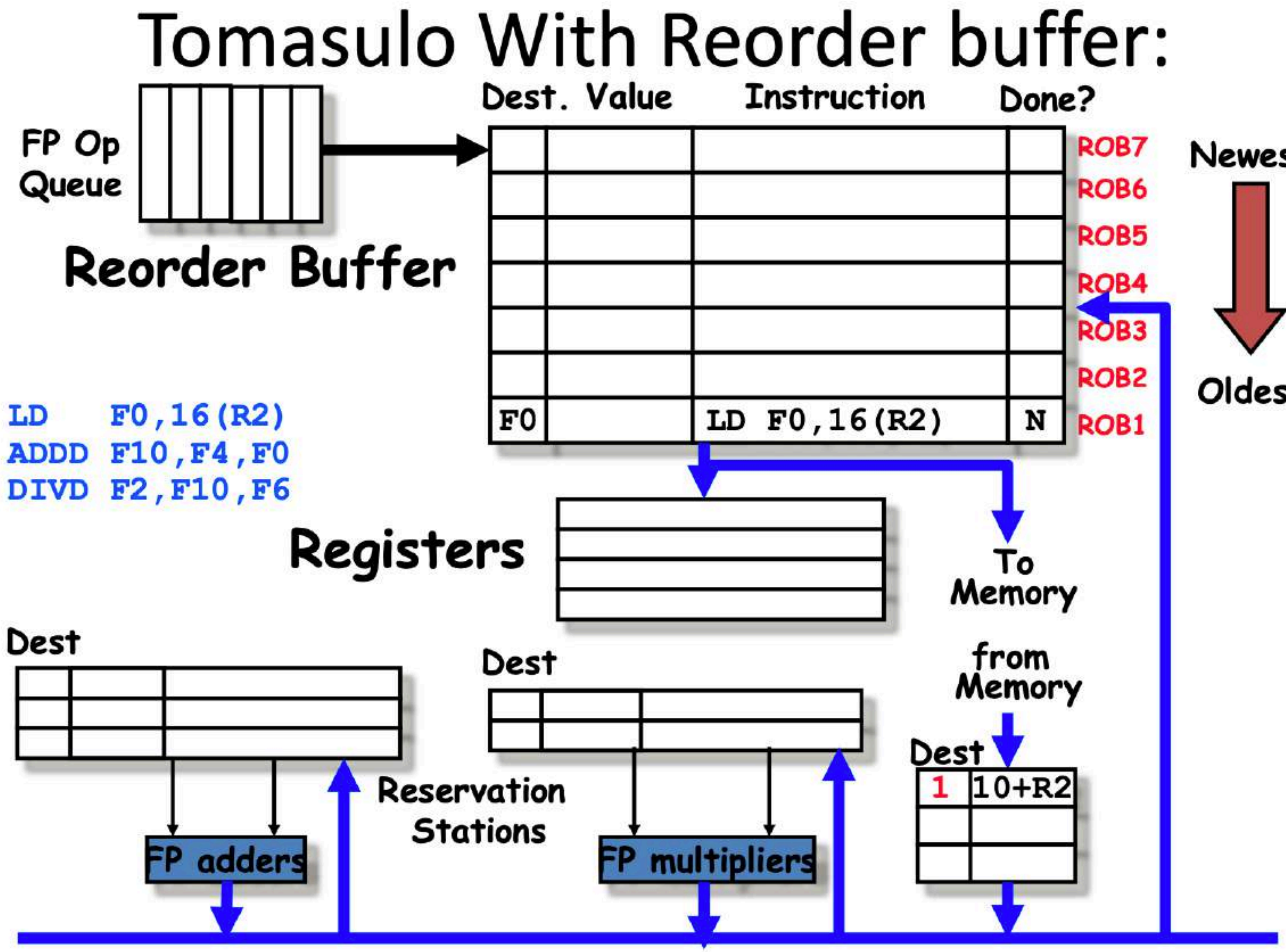
Reservation stations

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	

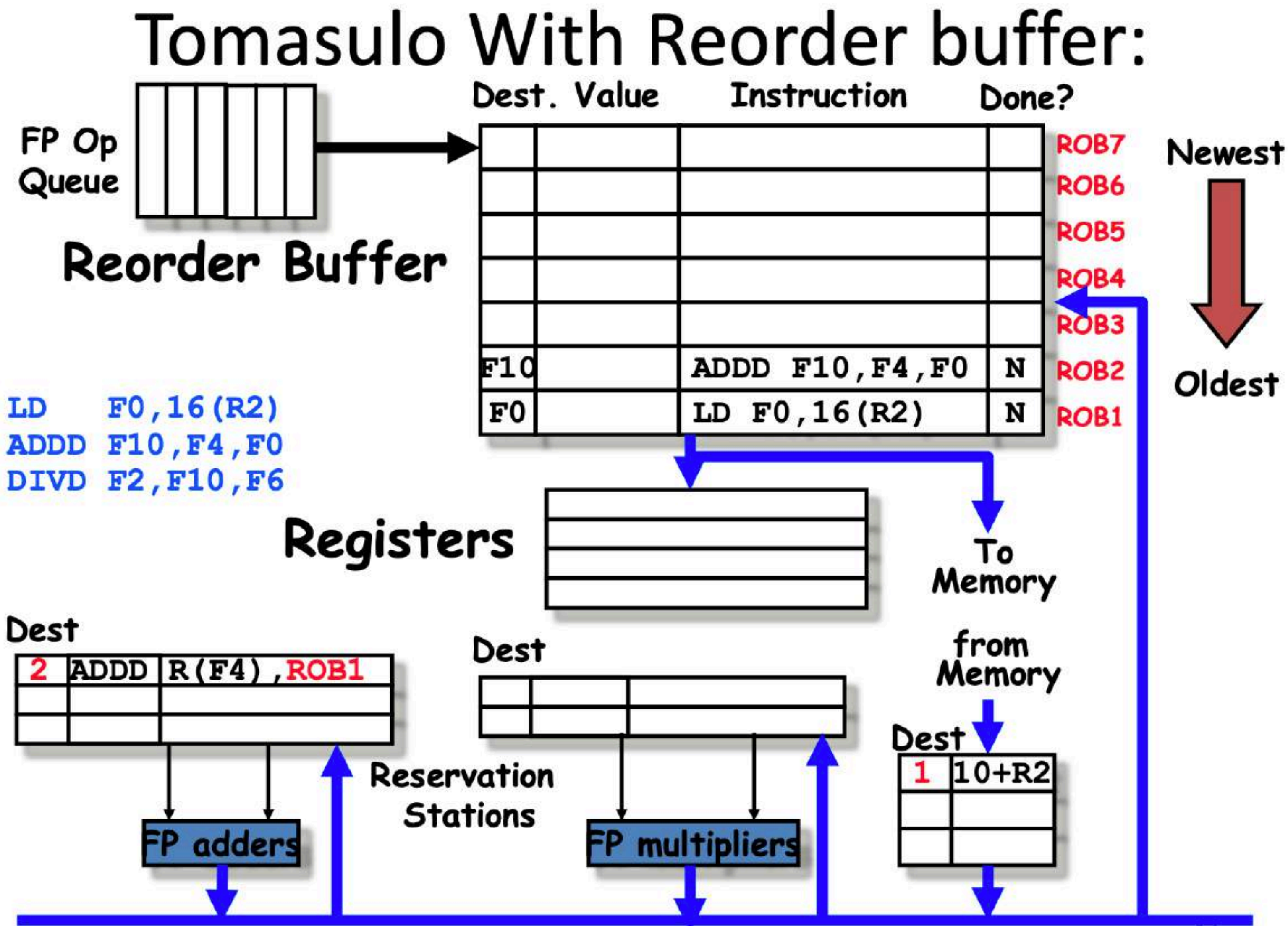
FP register status

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

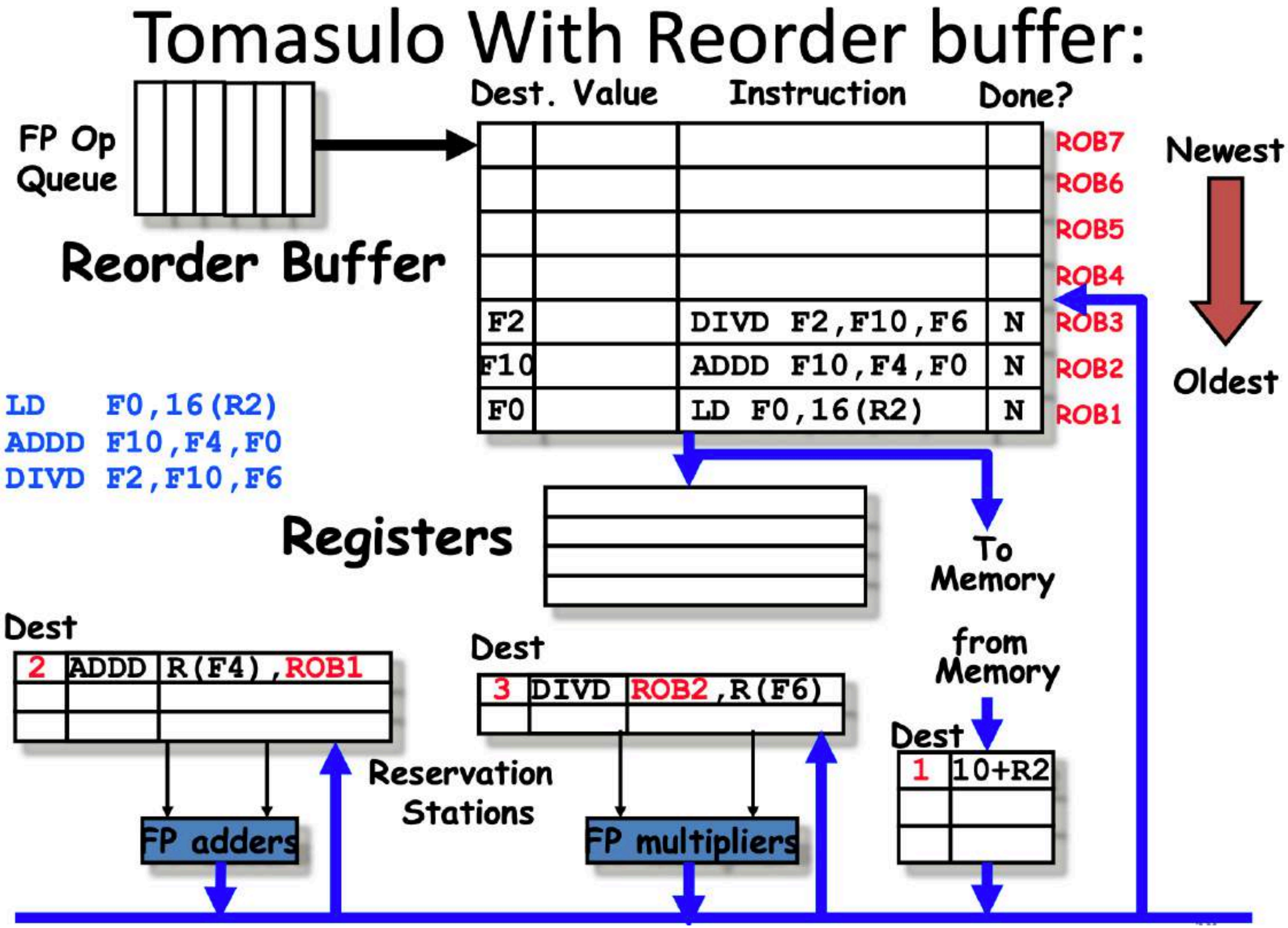
Reorder Buffer



Reorder Buffer



Reorder Buffer



Multiple Issue Processor

- To achieve CPI < 1, need to complete multiple instructions per clock
- Solutions:
  - Statically scheduled superscalar processors
  - VLIW (very long instruction word) processors
  - Dynamically scheduled superscalar processors

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

VLIW Processor

- Package multiple operations into one instruction
- Example VLIW processor:
  - One integer instruction (or branch)
  - Two independent floating-point operations
  - Two independent memory references

- Muse be enough parallelism in code to fill the available slots

VLIW Processor

Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation/branch
fld f0,0(x1)	fld f6,-8(x1)			
fld f10,-16(x1)	fld f14,-24(x1)			
fld f18,-32(x1)	fld f22,-40(x1)	fadd.d f4,f0,f2	fadd.d f8,f6,f2	
fld f26,-48(x1)		fadd.d f12,f0,f2	fadd.d f16,f14,f2	
		fadd.d f20,f18,f2	fadd.d f24,f22,f2	
fsd f4,0(x1)	fsd f8,-8(x1)	fadd.d f28,f26,f24		
fsd f12,-16(x1)	fsd f16,-24(x1)			addi x1,x1,-56
fsd f20,24(x1)	fsd f24,16(x1)			
fsd f28,8(x1)				bne x1,x2,Loop

- Disadvantages:
  - Statically finding parallelism
  - Code size
  - No hazard detection hardware
  - Binary code compatibility