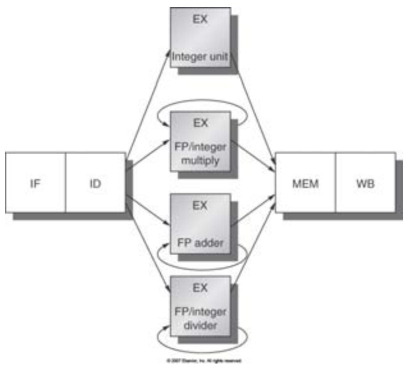
Diversified Pipelines—The Path Toward Superscalar Processors

HPCA, Spring 2011

Limitations of Our Simple 5-stage Pipeline

- Assumes single cycle EX stage for all instructions
- This is not feasible for
  - Complex integer operations
    - Multiply
    - Divide
    - Shift (possibly)
  - Floating Point Operations

A Naïve Extension of the 5 Stage Pipeline



Multicycle ALU Operations

- Latency: # of intervening cycles between the instruction that produced a result and a subsequent instruction that uses it.
- Initiation Interval: # of cycles between two instructions that utilize the same functional unit.

Functional Unit	Latency	Initiation Interval
Integer ALU	0	1
FP Add	3	1
Int Multiply	6	1
FP Multiply	6	1
FP Divide	24	25

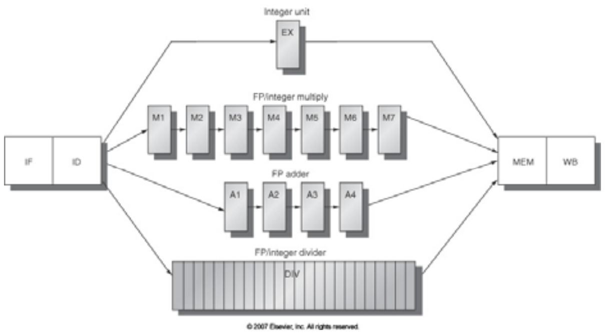
Multicycle ALU Operations

- Latency: # of intervening cycles between the instruction that produced a result and a subsequent instruction that uses it.
- Initiation Interval: # of cycles between two instructions that utilize the same functional unit.

Functional Unit	Latency	Initiation Interval	
Integer ALU	0	1	
FP Add	3	1	4 stage pipelined adder
Int Multiply	6	1	7 stage pipelined mult.
FP Multiply	6	1	
FP Divide	24	25	24 cycle divider (non-pipelined)

5

Diversified Pipeline



6

Problems with Diversified Pipeline

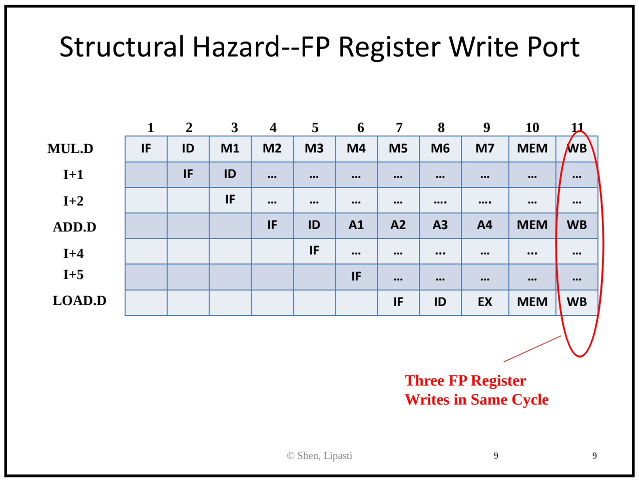
- Many more RAW hazard opportunities due to longer fp instruction execution times
- New Structural Hazards:
  - Divide instructions at distance < 25 (Due to non-pipelined Divide Unit.
  - Multiple Register Writes/Cycle due to variable instruction execution times
- Out-of-order instruction completion—Why is this a problem?
- WAW Hazards are possible (WAR not possible. Why?)

7

Structural Hazard--FP Register Write Port

	1	2	3	4	5	6	7	8	9	10	11
MUL.D	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
I+1		IF	ID	...	...	...	...	...	...	...	...
I+2			IF	...	...	...	...	....	....	...	...
ADD.D				IF	ID	A1	A2	A3	A4	MEM	WB
I+4					IF	...	...	...	...	...	...
I+5						IF	...	...	...	...	...
LOAD.D							IF	ID	EX	MEM	WB

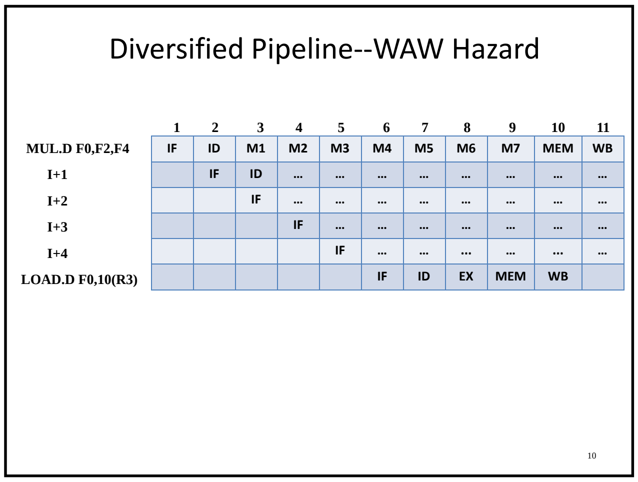
8



Structural Hazard--FP Register Write Port

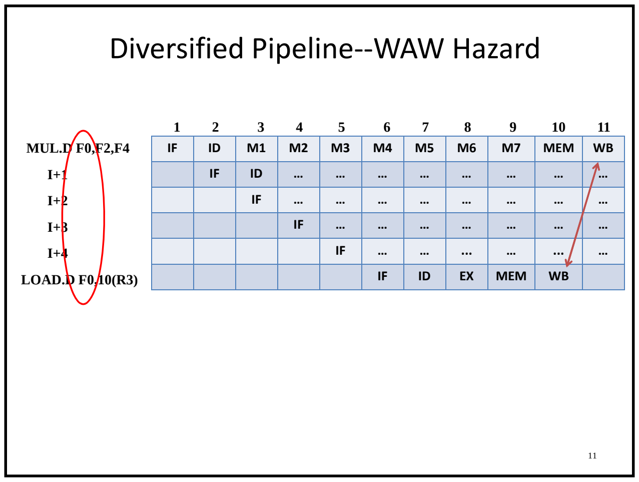
	1	2	3	4	5	6	7	8	9	10	11
MUL.D	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
I+1		IF	ID	…	…	…	…	…	…	…	…
I+2			IF	…	…	…	…	….	….	…	…
ADD.D				IF	ID	A1	A2	A3	A4	MEM	WB
I+4					IF	…	…	…	…	…	…
I+5						IF	…	…	…	…	…
LOAD.D							IF	ID	EX	MEM	WB

Three FP Register Writes in Same Cycle



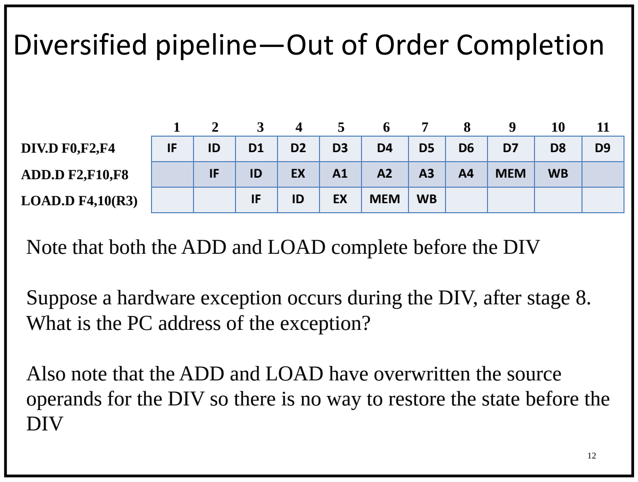
Diversified Pipeline--WAW Hazard

	1	2	3	4	5	6	7	8	9	10	11
MUL.D F0,F2,F4	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
I+1		IF	ID	…	…	…	…	…	…	…	…
I+2			IF	…	…	…	…	…	…	…	…
I+3				IF	…	…	…	…	…	…	…
I+4					IF	…	…	…	…	…	…
LOAD.D F0,10(R3)						IF	ID	EX	MEM	WB	



Diversified Pipeline--WAW Hazard

	1	2	3	4	5	6	7	8	9	10	11
MUL.D F0,F2,F4	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
I+1		IF	ID	…	…	…	…	…	…	…	…
I+2			IF	…	…	…	…	…	…	…	…
I+3				IF	…	…	…	…	…	…	…
I+4					IF	…	…	…	…	…	…
LOAD.D F0,10(R3)						IF	ID	EX	MEM	WB	



Diversified pipeline—Out of Order Completion

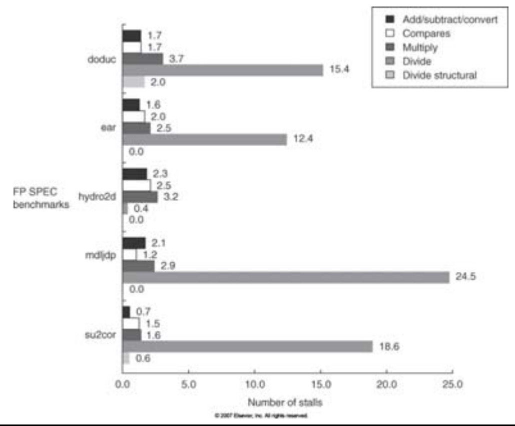
	1	2	3	4	5	6	7	8	9	10	11
DIV.D F0,F2,F4	IF	ID	D1	D2	D3	D4	D5	D6	D7	D8	D9
ADD.D F2,F10,F8		IF	ID	EX	A1	A2	A3	A4	MEM	WB	
LOAD.D F4,10(R3)			IF	ID	EX	MEM	WB				

Note that both the ADD and LOAD complete before the DIV

Suppose a hardware exception occurs during the DIV, after stage 8. What is the PC address of the exception?

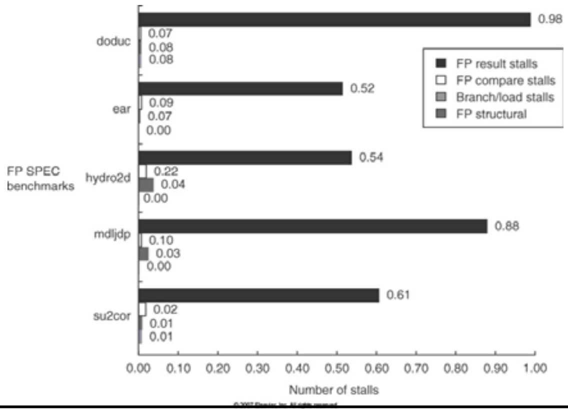
Also note that the ADD and LOAD have overwritten the source operands for the DIV so there is no way to restore the state before the DIV

Diversified Pipeline Performance



13

Diversified Pipeline—Stalls per Instruction



Diversified Pipeline—Can the Compiler Help?

- Consider this code to add a scalar to a vector:
```
for (i=1000; i>0; i=i–1)
 x[i] = x[i] + s;
```
- First translate into MIPS code:
  - -To simplify, assume 8 is lowest address

```
Loop: L.D    F0,0(R1);F0=vector element
             ADD.D    F4,F0,F2 ;add scalar from F2
             S.D      0(R1),F4 ;store result
             DADDUI   R1,R1,-8 ;decrement pointer 8B (DW)
             BNEZ     R1,Loop  ;branch R1!=zero
```

15

Can the Compiler Help?

```
Loop: L.D    F0,0(R1);F0=vector element
      ADD.D  F4,F0,F2;add scalar from F2
      S.D    0(R1),F4;store result
      DADDUI R1,R1,-8;decrement pointer 8B (DW)
      BNEZ   R1,Loop ;branch R1!=zero
```

- Assume the following pipeline latencies:
  - Ignore delayed branch in these examples

Instruction producing result	***Instruction using result***	***stalls between in cycles***
FP ADD	Another FP ALU op	3
FP ADD	Store double	2
Load double	FP ALU op	1
Load double	Store double	0
Integer op	Integer op	0

16

Stalls (NOPs) needed to account for Pipeline Latencies

```
1 Loop: L.D    F0,0(R1) ;F0=vector element
2       stall
3       ADD.D  F4,F0,F2 ;add scalar in F2
4       stall
5       stall
6       S.D    0(R1),F4 ;store result
7       DADDUI R1,R1,-8 ;decrement pointer 8B (DW)
8       stall           ;assumes can't forward to branch
9       BNEZ   R1,Loop  ;branch R1!=zero
```

Instruction producing result	*Instruction using result*	*Latency in clock cycles*
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1

- 9 clock cycles per loop iteration
- Can the compiler reorganize the code to minimize stalls?

17

Reorganized Code to Reduce Stalls

Swap DADDUI and S.D by changing address of S.D:

```
1 Loop: L.D    F0,0(R1)
2       DADDUI R1,R1,-8
3       ADD.D  F4,F0,F2
4       stall
5       stall
6       S.D    8(R1),F4 ;altered offset when move DADUI
7       BNEZ   R1,Loop
```

Instruction producing result	*Instruction using result*	*Latency in clock cycles*
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1

7 clock cycles, but just 3 for execution (L.D, ADD.D,S.D), 4 for loop overhead; Can we(the compiler) do better?

18

Loop Unrolling to Improve Performance

```
1 Loop:L.D    F0,0(R1)      1 cycle stall
3      ADD.D  F4,F0,F2      2 cycles stall
6      S.D    0(R1),F4     ;drop DADDUI & BNEZ
7      L.D    F6,-8(R1)
9      ADD.D  F8,F6,F2
12     S.D    -8(R1),F8    ;drop DADDUI & BNEZ
13     L.D    F10,-16(R1)
15     ADD.D  F12,F10,F2
18     S.D    -16(R1),F12  ;drop ADDUI & BNEZ
19     L.D    F14,-24(R1)
21     ADD.D  F16,F14,F2
24     S.D    -24(R1),F16
25     DADDUI R1,R1,#-32   ;alter to 4*8
26     BNEZ   R1,LOOP
```

27 clock cycles, or 6.75 per iteration

(Assumes R1 is multiple of 4)

19

Loop Unrolling with Code Rearrangement

```
1 Loop:L.D    F0,0(R1)
2      L.D    F6,-8(R1)
3      L.D    F10,-16(R1)
4      L.D    F14,-24(R1)
5      ADD.D  F4,F0,F2
6      ADD.D  F8,F6,F2
7      ADD.D  F12,F10,F2
8      ADD.D  F16,F14,F2
9      S.D    0(R1),F4
10     S.D    -8(R1),F8
11     S.D    -16(R1),F12
12     DSUBUI R1,R1,#32
13     S.D    8(R1),F16 ; 8-32 = -24
14     BNEZ   R1,LOOP
```

14 clock cycles, or 3.5 per iteration

20

Limits to Loop Unrolling

1. Decrease in amount of overhead amortized with each extra unrolling
   - Amdahl's Law
2. Growth in code size
   - For larger loops, concern it increases the instruction cache miss rate
3. Register pressure: potential shortfall in registers created by aggressive unrolling and scheduling
   - If not possible to allocate all live values to registers, may lose some or all of its advantage

- Loop unrolling reduces impact of branches on pipeline; another way is branch prediction

21

Hardware-based Performance Optimization-- Dynamic Scheduling

- Dynamic scheduling - hardware rearranges the instruction execution to reduce stalls while maintaining data flow and exception behavior
  - Handles cases when dependences unknown at compile time
  - Allows the processor to tolerate unpredictable delays such as cache misses, by executing other code while waiting for the miss to resolve
  - Allows code to be compiled independently of details of a particular pipeline
  - Simplifies the compiler
- Hardware speculation, a technique with significant performance advantages, builds on dynamic scheduling (more about this later)

22

Dynamic Scheduling Example

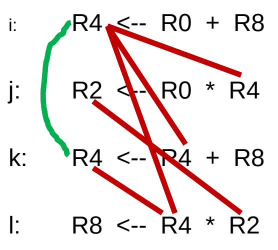
Consider:

```
i:    R4  <--  R0  +  R8

j:    R2  <--  R0  *  R4

k:    R4  <--  R4  +  R8

l:    R8  <--  R4  *  R2
```

23

Dynamic Scheduling Example

Consider:

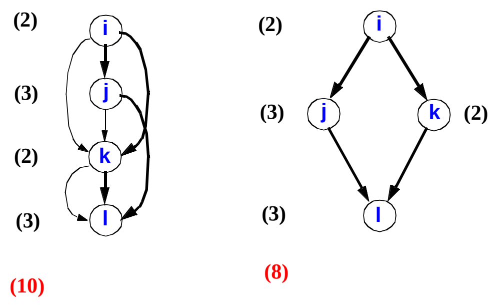


```
i:    R4  <--  R0  +  R8

j:    R2  <--  R0  *  R4

k:    R4  <--  R4  +  R8

l:    R8  <--  R4  *  R2
```

RAW Hazards WAW Hazards

24

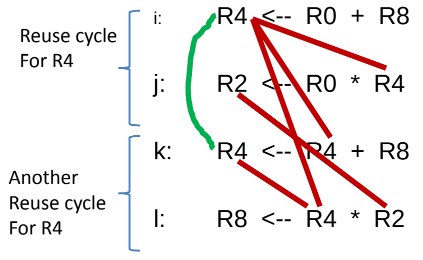
Dynamic Scheduling—The dataflow limit



Objective of Dynamic Scheduling is to come as close as possible of the Dataflow Limit

Dynamic Scheduling Example

Consider:



Reuse cycle For R4

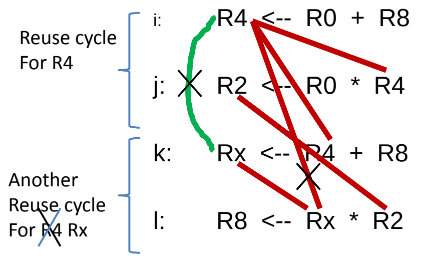
Another Reuse cycle For R4

```
i:  R4 <-- R0 + R8
j:  R2 <-- R0 * R4
k:  R4 <-- R4 + R8
l:  R8 <-- R4 * R2
```

RAW Hazards   WAW Hazards

Dynamic Scheduling Example

Consider:



Reuse cycle For R4

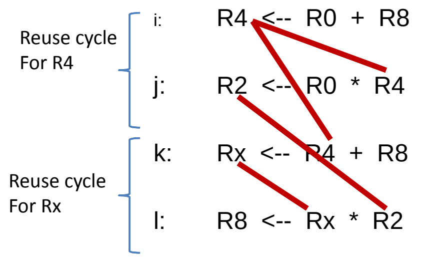
Another Reuse cycle For ~~R4~~ Rx

```
i:  R4 <-- R0 + R8
j:  R2 <-- R0 * R4
k:  Rx <-- R4 + R8
l:  R8 <-- Rx * R2
```

RAW Hazards   WAW Hazards

Dynamic Scheduling Example

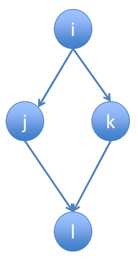
Consider:



Reuse cycle For R4

Reuse cycle For Rx

```
i:  R4 <-- R0 + R8
j:  R2 <-- R0 * R4
k:  Rx <-- R4 + R8
l:  R8 <-- Rx * R2
```



RAW Hazards

Dynamic Scheduling

- Key idea: Allow instruction(s) following a 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)
- Will distinguish when an instruction *begins execution* and when it *completes execution*; between these times, the instruction is *in execution*
- Note: Dynamic execution creates WAR and WAW hazards and makes exceptions harder

Dynamic Scheduling—Starting Point

- Split the ID pipe stage of simple 5-stage pipeline into 2 stages:
- *Issue*—Decode instructions, check for structural hazards
- *Read operands*—Wait until no data hazards, then read operands

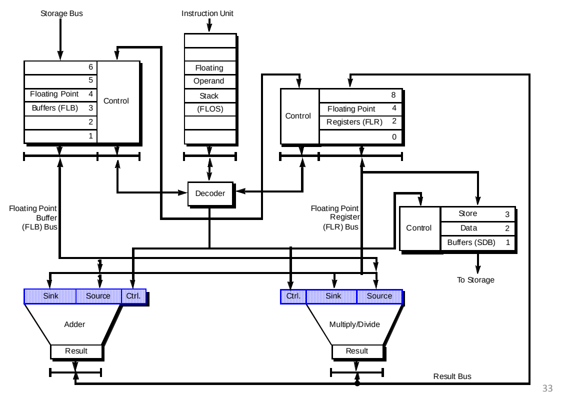
Dynamic Scheduling: Tomasulo's Algorithm

- For IBM 360/91 (late 1960s, before caches!)
  - ⇒ Long memory latency
- Goal: High Performance without special compilers
- Small number of floating point registers (4 in 360) prevented interesting compiler scheduling of operations
  - This led Tomasulo to try to figure out how to get more effective registers — renaming in hardware!
- Why Study 1966 Computer?
- Tomasulo's algorithm is the basis for dynamic scheduling approach used in most modern processors

Tomasulo's Algorithm

- Control & buffers distributed with Functional Units (FU)
  - FU buffers called "reservation stations"; have pending operands
- Registers in instructions replaced by values or pointers to reservation stations(RS); called register renaming ;
  - Renaming avoids WAR, WAW hazards
  - More reservation stations than registers, so can do optimizations compilers can't
- Result forwarding via a 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's Algorithm [Tomasulo, 1967]



33

IBM 360/91 FPU

- **Multiple functional units (FU's)**
  - Floating-point add
  - Floating-point multiply/divide
- **Three register files (pseudo reg-reg machine in floating-point unit)**
  - (4) floating-point registers (FLR)
  - (6) floating-point buffers (FLB)
  - (3) store data buffers (SDB)
- **Out of order instruction execution**:
  - After decode the instruction unit passes all floating point instructions (in order) to the floating-point operation stack (FLOS).
  - In the floating point unit, instructions are then further decoded and issued from the FLOS to the two FU's
- **Variable operation latencies**:
  - Floating-point add: 2 cycles
  - Floating-point multiply: 3 cycles
  - Floating-point divide: 12 cycles
- Goal: **achieve concurrent execution of multiple floating-point instructions, in addition to achieving one instruction per cycle in instruction pipeline**

34

Dependence Mechanisms

Two Address IBM 360 Instruction Format:

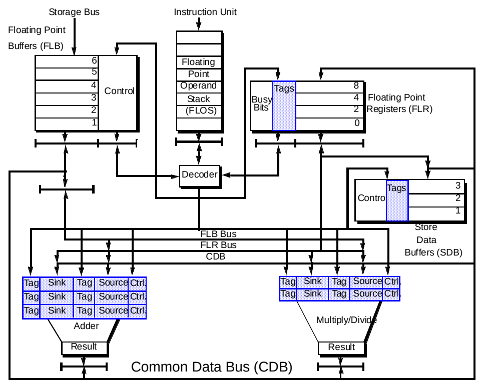
R1 <-- R1 op R2

Major dependence mechanisms:

- **Structural (FU) dependence = > virtual FU's**
  - Reservation stations
- **True dependence = > pseudo operands + result forwarding**
  - Register tags
  - Reservation stations
  - Common data bus (CDB)
- **Anti-dependence = > operand copying**
  - Reservation stations
- **Output dependence = > register renaming + result forwarding**
  - Register tags
  - Reservation stations
  - Common data bus (CDB)

35

IBM 360/91 FPU



36

Reservation Stations

- Used to collect operands or pseudo operands (tags).
- Associate more than one set of buffering registers (control, source, sink) with each FU, = > virtual FU's.
- Add unit: three reservation stations
- Multiply/divide unit: two reservation stations

Common Data Bus (CDB)

- **CDB is fed by all units that can alter a register (or supply register values) and it feeds all units which can have a register as an operand**.
- Sources of CDB:
  - Floating-point buffers (FLB)
  - Two FU's (add unit and the multiply/divide unit)
- Destinations of CDB:
  - Reservation stations
  - Floating-point registers (FLR)
  - Store data buffers (SDB)

Register Tags

- **Every source of a register value must be uniquely identified by its own tag value.**
  - (6) FLB's
  - (5) reservation stations (3 with add unit, 2 with multiply/divide unit)

    = = > 4-bit tag is needed to identify the 11 potential sources
- **Every destination of a register value must carry a tag field**.
  - (5) "sink" entries of the reservation stations
  - (5) "source" entries of the reservation stations
  - (4) FLR's
  - (3) SDB's

    = = > a total of 17 tag fields are needed (i.e. 17 places that need tags)

Operation of Dependence Mechanisms

1. **Structural (FU) dependence = > virtual FU's**
   - FLOS can hold and decode up to 8 instructions.
   - Instructions are dispatched to the 5 reservation stations (virtual FU's) even though there are only two physical FU's.
   - Hence, structural dependence does not stall dispatching.
2. **True dependence = > pseudo operands + result forwarding**
   - If an operand is available in FLR, it is copied to a res. station entry.
   - If an operand is not available (i.e. there is pending write), then a tag is copied to the reservation station entry instead. This tag identifies the source of the pending write. This instruction then waits in its reservation station for the true dependence to be resolved.
   - When the operand is finally produced by the source (ID of source = tag value), this source unit asserts its ID, i.e. its tag value, on the CDB followed by broadcasting of the operand on the CDB.
   - All the reservation station entries and the FLR entries and SDB entries carrying this tag value in their tag fields will detect a match of tag values and latch in the broadcasted operand from the CDB.
   - Hence, true dependence does not block subsequent independent instructions and does not stall a physical FU. Forwarding also minimizes delay due to true dependence.

Example 1

i: R2 <- R0 + R4

j: R8 <- R0 + R2

CYCLE #1

ID	Tag	Sink	Tag	Source
1				
2				
3				

Adder

ID	Tag	Sink	Tag	Source
4				
5				

Mult/Div

	Busy	Tag	Data
0			6.0
2			3.5
4			10.0
8			7.8

DISPATCHED INSTRUCTION(S): ______________

CYCLE #2

ID	Tag	Sink	Tag	Source
1				
2				
3				

Adder

ID	Tag	Sink	Tag	Source
4				
5				

Mult/Div

	Busy	Tag	Data
0			6.0
2			3.5
4			10.0
8			7.8

DISPATCHED INSTRUCTION(S): ______________

CYCLE #3

ID	Tag	Sink	Tag	Source
1				
2				
3				

Adder

ID	Tag	Sink	Tag	Source
4				
5				

Mult/Div

	Busy	Tag	Data
0			
2			
4			
8			

DISPATCHED INSTRUCTION(S): ______________

41

Operation of Dependence Mechanisms

3. Anti-dependence = > operand copying

- **If an operand is available in FLR, it is copied to a reservation station entry.**
- **By copying this operand to the reservation station, all anti-dependences due to future writes to this same register are resolved.**
- **Hence, the reading of an operand is not delayed, possibly due to other dependences, and subsequent writes are also not delayed.**

42

Example 2

i: R4 <- R0 * R8

j: R0 <- R4 * R2

k: R2 <- R2 + R8

CYCLE #1

ID	Tag	Sink	Tag	Source
1				
2				
3				

Adder

ID	Tag	Sink	Tag	Source
4				
5				

Mult/Div

	Busy	Tag	Data
0			6.0
2			3.5
4			10.0
8			7.8

DISPATCHED INSTRUCTION(S): ______________

CYCLE #2

ID	Tag	Sink	Tag	Source
1				
2				
3				

Adder

ID	Tag	Sink	Tag	Source
4				
5				

Mult/Div

	Busy	Tag	Data
0			6.0
2			3.5
4			10.0
8			7.8

DISPATCHED INSTRUCTION(S): ______________

CYCLE #3

ID	Tag	Sink	Tag	Source
1				
2				
3				

Adder

ID	Tag	Sink	Tag	Source
4				
5				

Mult/Div

	Busy	Tag	Data
0			
2			
4			
8			

DISPATCHED INSTRUCTION(S): ______________

43

Operation of Dependence Mechanisms

3. Output dependence = > register renaming + result forwarding

- **If a register is waiting for a pending write, its tag field will contain the ID, or tag value, of the source for that pending write.**
- **When that source eventually produces the result, that result will be written into the register via the CDB.**
- **It is possible that prior to the completion of the pending write, another instruction can come along and also has that same register as its destination register.**
- **If this occurs, the operands (or pseudo operands) needed by this instruction are still copied to an available reservation station. In addition, the tag field of the destination register of this instruction is updated with the ID of this new reservation station, i.e. the old tag value is overwritten. This will ensure that the said register will get the latest value, i.e. the late completing earlier write cannot overwrite a later write.**
- **Hence, the output dependence is resolved without stalling a physical functional unit, not requiring additional buffers to ensure sequential write back to the register file.**

44

Example 3

i: R4 <- R0 * R8

j: R2 <- R0 + R4

k: R4 <- R0 + R8

l: R8 <- R4 * R8

CYCLE #1

ID	Tag	Sink	Tag	Source
1				
2				
3				

Adder

ID	Tag	Sink	Tag	Source
4				
5				

Mult/Div

	Busy	Tag	Data
0			6.0
2			3.5
4			10.0
8			7.8

DISPATCHED INSTRUCTION(S): _______________

CYCLE #2

ID	Tag	Sink	Tag	Source
1				
2				
3				

Adder

ID	Tag	Sink	Tag	Source
4				
5				

Mult/Div

	Busy	Tag	Data
0			6.0
2			3.5
4			10.0
8			7.8

DISPATCHED INSTRUCTION(S): _______________

CYCLE #3

ID	Tag	Sink	Tag	Source
1				
2				
3				

Adder

ID	Tag	Sink	Tag	Source
4				
5				

Mult/Div

	Busy	Tag	Data
0			
2			
4			
8			

DISPATCHED INSTRUCTION(S): _______________

Summary of Tomasulo's Algorithm

- **Supports out of order execution of instructions.**
- **Resolves dependences dynamically using hardware.**
- **Attempts to delay the resolution of dependencies as late as possible.**
- **Structural dependence does not stall issuing; virtual FU's in the form of reservation stations are used.**
- **Output dependence does not stall issuing; copying of old tag to reservation station and updating of tag field of the register with pending write with the new tag.**
- **True dependence with a pending write operand does not stall the reading of operands; pseudo operand (tag) is copied to reservation station.**
- **Anti-dependence does not stall write back; earlier copying of operand awaiting read to the reservation station.**
- **Can support sequence of multiple output dependences.**
- **Forwarding from FU's to reservation stations bypasses the register file.**

Example 4

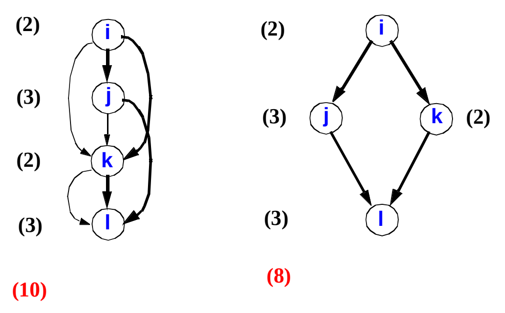
i: R4 <-- R0 + R8

j: R2 <-- R0 * R4

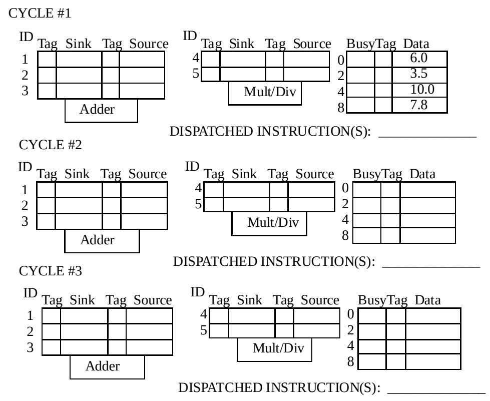
k: R4 <-- R4 + R8

l: R8 <-- R4 * R2

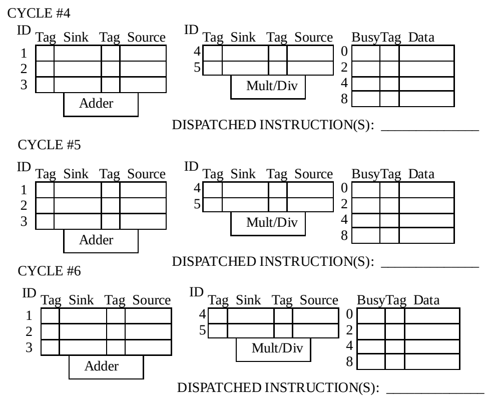
Example 4



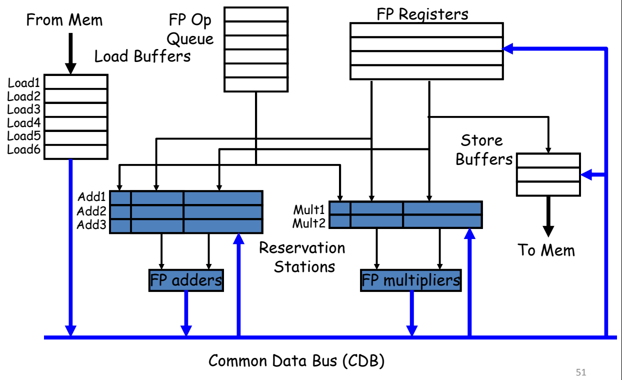
Example 4



Example 4



Tomasulo Revisited—H &P notation



Reservation Station Components

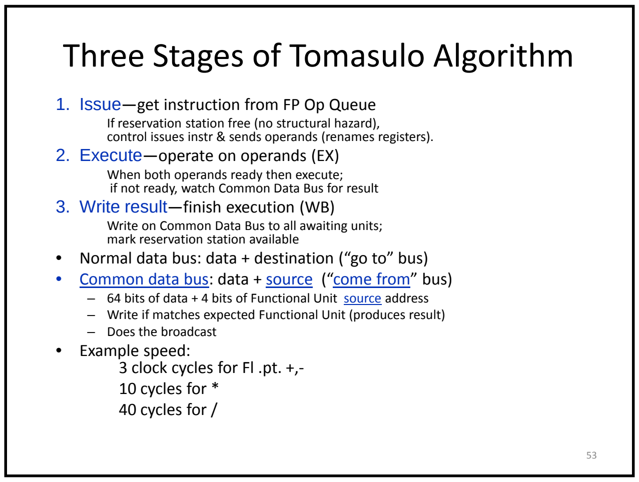
Op: Operation to perform in the unit (e.g., + or –)

Vj, Vk: Value of Source operands
- Store buffers has V field, result to be stored

Qj, Qk: Reservation stations producing source registers (value to be written)
- Note: Qj,Qk=0 => ready
- Store buffers only have Qi for RS producing result

Busy: Indicates reservation station or FU is busy

Register result status—Indicates which functional unit will write each register, if one exists. Blank when no pending instructions that will write that register.

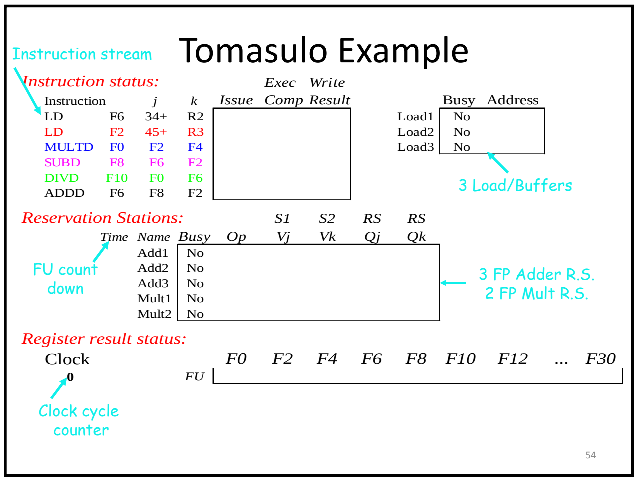


Three Stages of Tomasulo Algorithm

1. Issue—get instruction from FP Op Queue
   If reservation station free (no structural hazard), control issues instr & sends operands (renames registers).
2. Execute—operate on operands (EX)
   When both operands ready then execute; if not ready, watch Common Data Bus for result
3. Write result—finish execution (WB)
   Write on Common Data Bus to all awaiting units; mark reservation station available

- Normal data bus: data + destination ("go to" bus)
- Common data bus: data + source ("come from" bus)
  - 64 bits of data + 4 bits of Functional Unit source address
  - Write if matches expected Functional Unit (produces result)
  - Does the broadcast
- Example speed:
  3 clock cycles for Fl .pt. +,-
  10 cycles for *
  40 cycles for /

53



Tomasulo Example

Instruction stream

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2			
LD	F2	45+	R3			
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

3 Load/Buffers

Reservation Stations:

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

FU count down

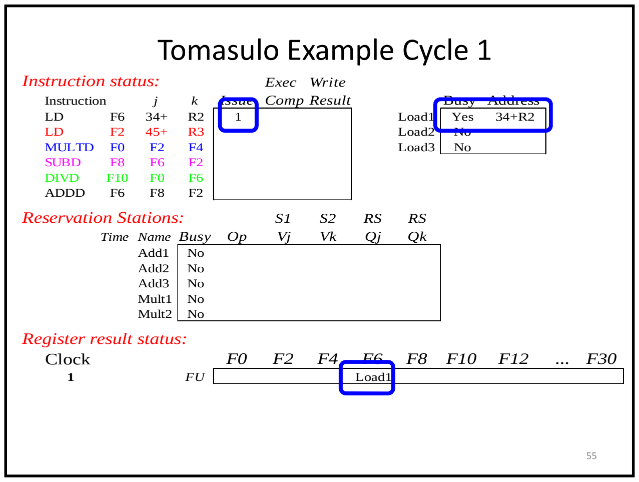
3 FP Adder R.S.
2 FP Mult R.S.

Register result status:

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

Clock cycle counter

54



Tomasulo Example Cycle 1

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1		
LD	F2	45+	R3			
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	No	
Load3	No	

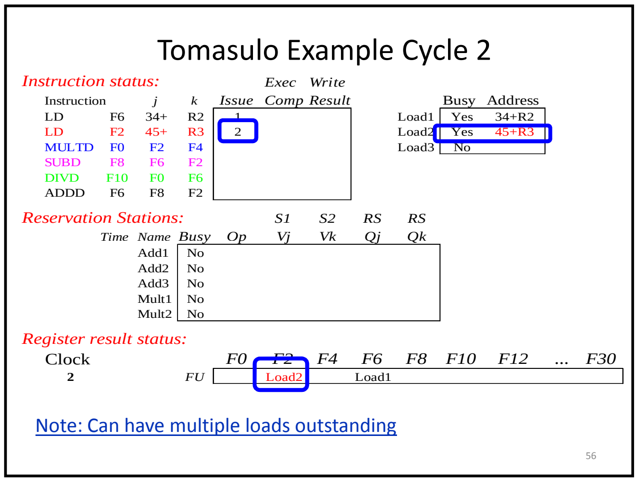
Reservation Stations:

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

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
1	*FU*				Load1					

55



Tomasulo Example Cycle 2

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1		
LD	F2	45+	R3	2		
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	Yes	45+R3
Load3	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					

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
2	*FU*		Load2		Load1					

Note: Can have multiple loads outstanding

56

Tomasulo Example Cycle 3

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	
LD	F2	45+	R3	2		
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	Yes	45+R3
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	MULTD		R(F4)	Load2	
	Mult2	No					

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
3	FU	Mult1	Load2		Load1					

- Note: registers names are removed ("renamed") in Reservation Stations; MULT issued
- Load1 completing; what is waiting for Load1?

57

Tomasulo Example Cycle 4

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	Yes	45+R3
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	Yes	SUBD	M(A1)			Load2
	Add2	No					
	Add3	No					
	Mult1	Yes	MULTD		R(F4)	Load2	
	Mult2	No					

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
4	FU	Mult1	Load2		M(A1)	Add1				

- Load2 completing; what is waiting for Load2?

58

Tomasulo Example Cycle 5

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
2	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	No					
	Add3	No					
10	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
5	FU	Mult1	M(A2)		M(A1)	Add1	Mult2			

- Timer starts down for Add1, Mult1

59

Tomasulo Example Cycle 6

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

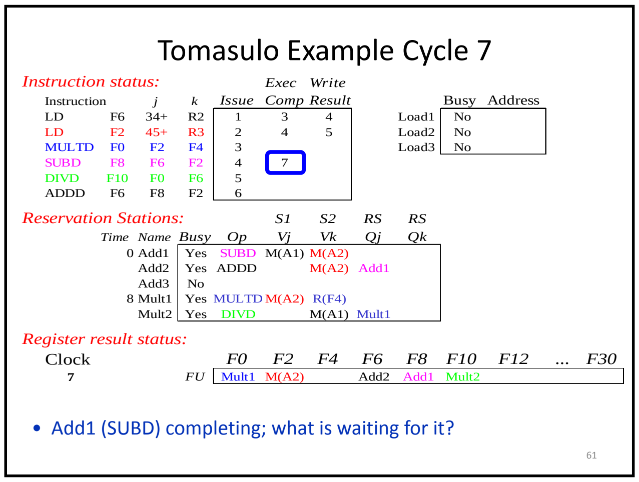
Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
1	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	Yes	ADDD		M(A2)	Add1	
	Add3	No					
9	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
6	FU	Mult1	M(A2)		Add2	Add1	Mult2			

- Issue ADDD here despite name dependency on F6?

60



Tomasulo Example Cycle 7

Instruction status:

Instruction	j	k	Issue	Exec Comp	Write Result
LD F6	34+	R2	1	3	4
LD F2	45+	R3	2	4	5
MULTD F0	F2	F4	3		
SUBD F8	F6	F2	4	7	
DIVD F10	F0	F6	5		
ADDD F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

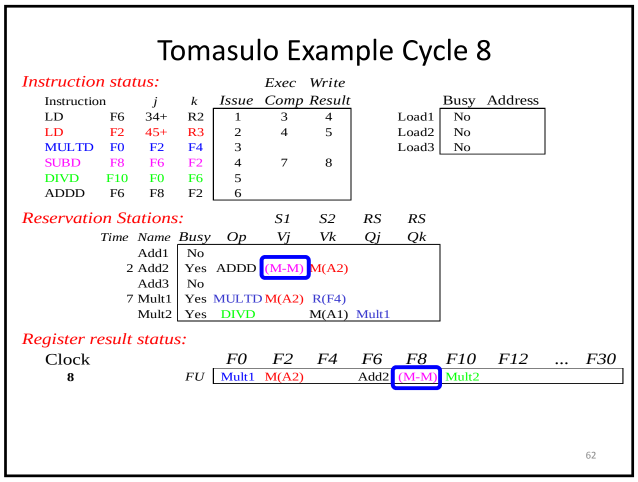
Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
0	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	Yes	ADDD		M(A2)	Add1	
	Add3	No					
8	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
7	FU	Mult1	M(A2)		Add2	Add1	Mult2			

- Add1 (SUBD) completing; what is waiting for it?

61



Tomasulo Example Cycle 8

Instruction status:

Instruction	j	k	Issue	Exec Comp	Write Result
LD F6	34+	R2	1	3	4
LD F2	45+	R3	2	4	5
MULTD F0	F2	F4	3		
SUBD F8	F6	F2	4	7	8
DIVD F10	F0	F6	5		
ADDD F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

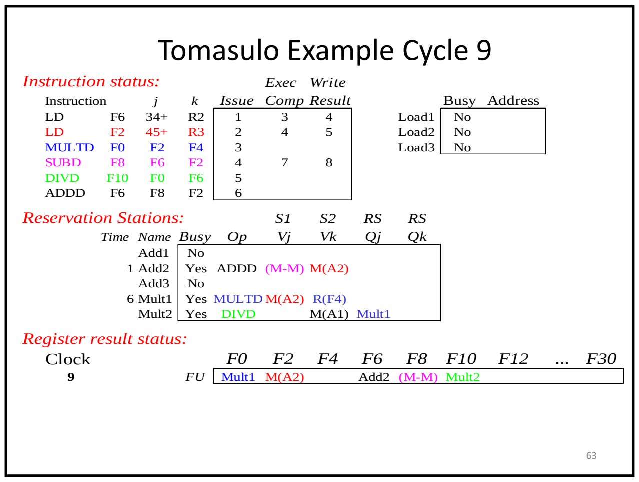
Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
2	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
7	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
8	FU	Mult1	M(A2)		Add2	(M-M)	Mult2			

62



Tomasulo Example Cycle 9

Instruction status:

Instruction	j	k	Issue	Exec Comp	Write Result
LD F6	34+	R2	1	3	4
LD F2	45+	R3	2	4	5
MULTD F0	F2	F4	3		
SUBD F8	F6	F2	4	7	8
DIVD F10	F0	F6	5		
ADDD F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

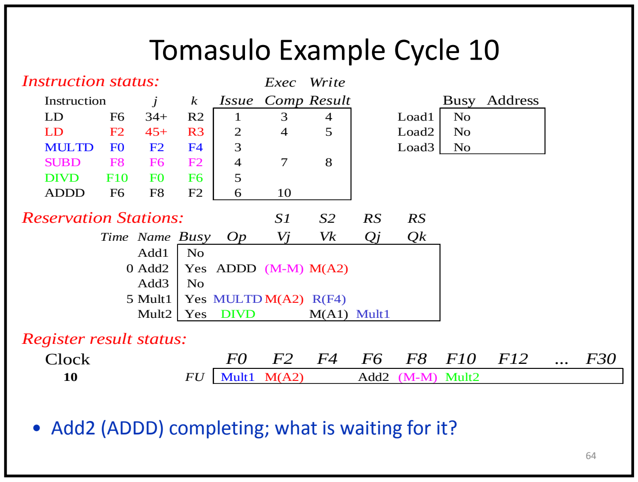
Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
1	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
6	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
9	FU	Mult1	M(A2)		Add2	(M-M)	Mult2			

63



Tomasulo Example Cycle 10

Instruction status:

Instruction	j	k	Issue	Exec Comp	Write Result
LD F6	34+	R2	1	3	4
LD F2	45+	R3	2	4	5
MULTD F0	F2	F4	3		
SUBD F8	F6	F2	4	7	8
DIVD F10	F0	F6	5		
ADDD F6	F8	F2	6	10	

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

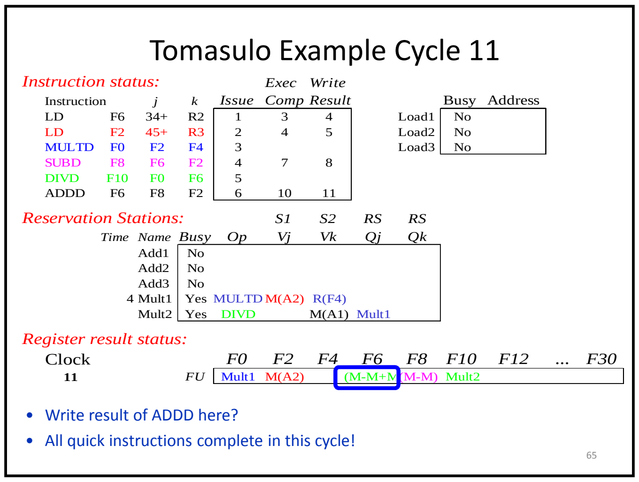
Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
0	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
5	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
10	FU	Mult1	M(A2)		Add2	(M-M)	Mult2			

- Add2 (ADDD) completing; what is waiting for it?

64



Tomasulo Example Cycle 11

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

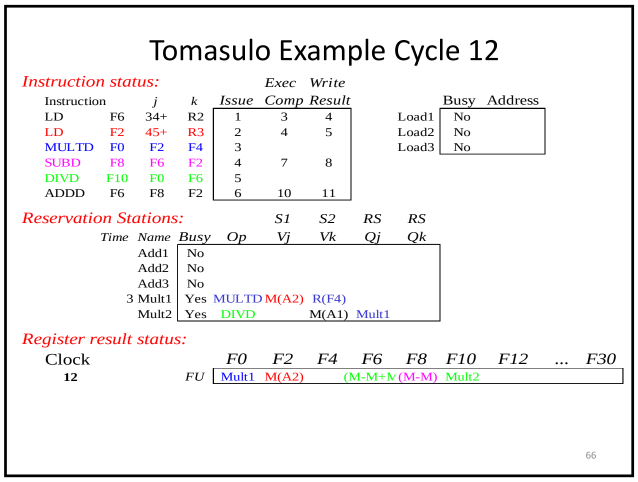
Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
4	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
11	FU	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

- Write result of ADDD here?
- All quick instructions complete in this cycle!

65



Tomasulo Example Cycle 12

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

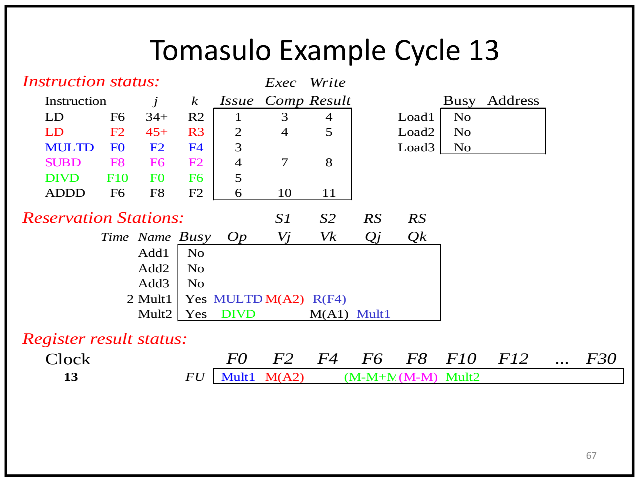
Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
3	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
12	FU	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

66



Tomasulo Example Cycle 13

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

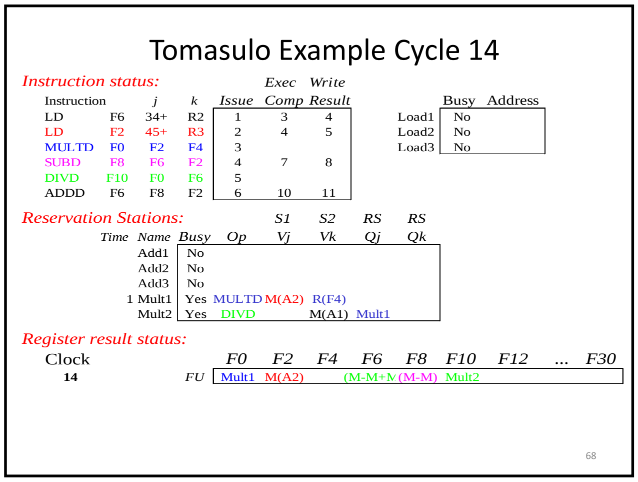
Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
2	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
13	FU	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

67



Tomasulo Example Cycle 14

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

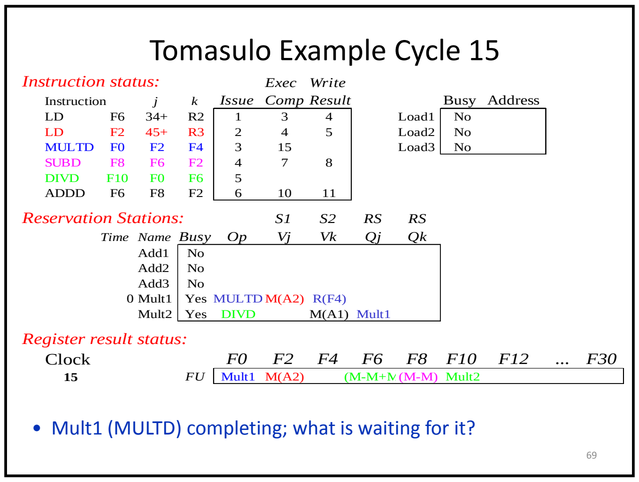
Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
1	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
14	FU	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

68



Tomasulo Example Cycle 15

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

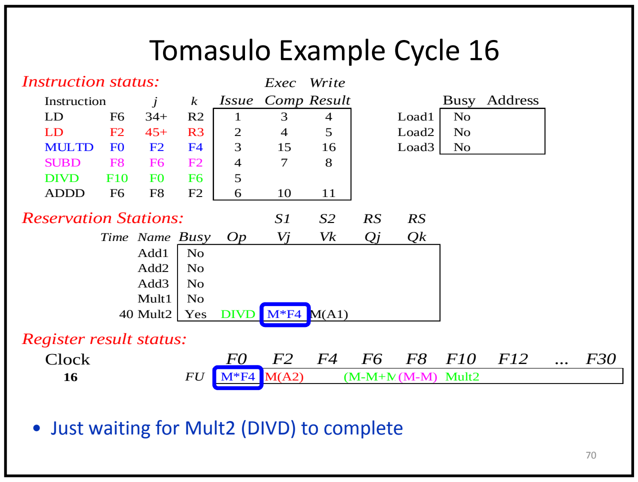
Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
0	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
15	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

- Mult1 (MULTD) completing; what is waiting for it?

69



Tomasulo Example Cycle 16

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

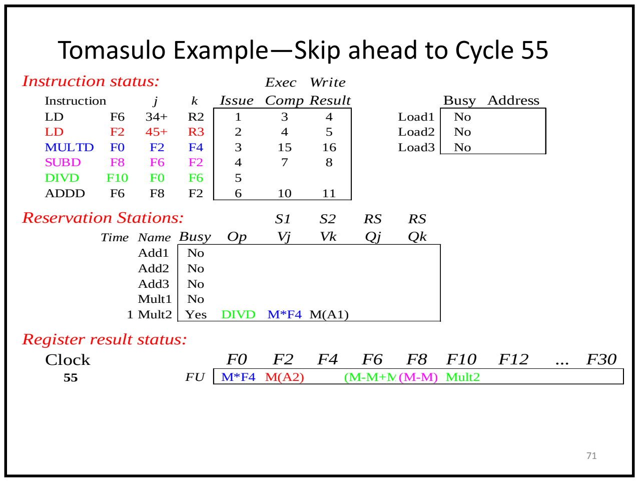
Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
40	Mult2	Yes	DIVD	$\boxed{\text{M*F4}}$	M(A1)		

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
16	*FU*	$\boxed{\text{M*F4}}$	M(A2)		(M-M+M	(M-M)	Mult2			

- Just waiting for Mult2 (DIVD) to complete

70



Tomasulo Example—Skip ahead to Cycle 55

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

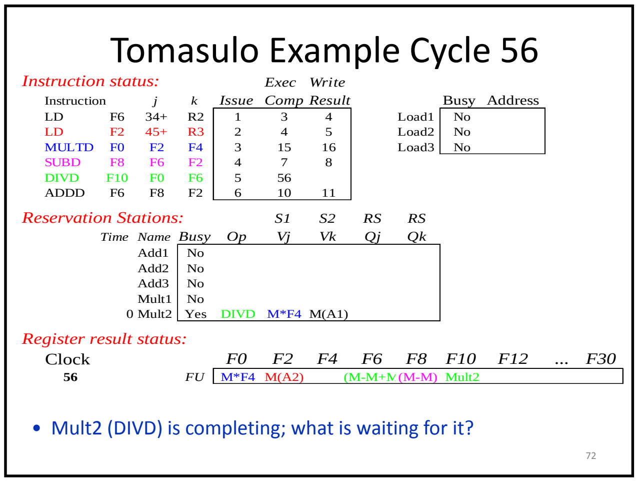
Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
1	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
55	*FU*	M*F4	M(A2)		(M-M+M	(M-M)	Mult2			

71



Tomasulo Example Cycle 56

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5	56	
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
0	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
56	*FU*	M*F4	M(A2)		(M-M+M	(M-M)	Mult2			

- Mult2 (DIVD) is completing; what is waiting for it?

72

Tomasulo Example Cycle 57

Instruction status:

Instruction	j	k	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5	56	57
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
56	*FU*	M*F4	M(A2)		(M-M+M	(M-M)	Result			

- Once again: In-order issue, out-of-order execution and out-of-order completion.

73

Why can Tomasulo overlap iterations of loops?

- 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
- Other perspective: Tomasulo building data flow dependency graph on the fly

74

Tomasulo's scheme offers Two major 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

75

Tomasulo Drawbacks

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

76

Dynamic Scheduling--Conclusions

- Leverage Implicit Parallelism for Performance: Instruction Level Parallelism
- Loop unrolling by compiler to increase ILP
- Branch prediction to increase ILP
- Dynamic HW exploiting ILP
  - Works when can't know dependence at compile time
  - Can hide L1 cache misses
  - Code for one machine runs well on another

77

Dynamic Scheduling—Conclusions (cont.)

- Reservations stations: *renaming* to larger set of registers + buffering source operands
  - Prevents registers as bottleneck
  - Avoids WAR, WAW hazards
  - Allows loop unrolling in HW
- Not limited to basic blocks
  (integer unit gets ahead, beyond branches)
- Helps cache misses as well
- Lasting Contributions
  - Dynamic scheduling
  - Register renaming
  - Load/store disambiguation
- 360/91 descendants are Intel Pentium 4, IBM Power 5, AMD Athlon/Opteron, …

78

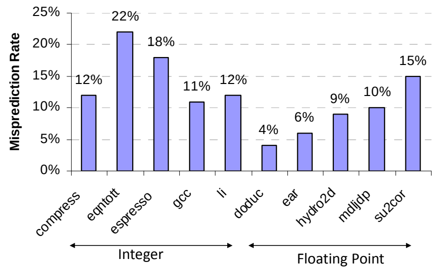
Performance Enhancement—Better Branch Prediction

- Accurate Branch Prediction becomes more important with dynamic scheduling
  - Dynamic scheduling may stall if it can't look past branch points
  - Cost of misprediction may be high

Static Branch Prediction

- Earlier, we discussed scheduling code around delayed branch
- To reorder code around branches, need to predict branch statically at compile time
- Simplest scheme is to predict a branch as taken
  - Average misprediction = untaken branch frequency = 34% SPEC

• More accurate scheme predicts branches using profile information collected from earlier runs, and modify prediction based on last run:



80

Dynamic (Run-time) Branch Prediction

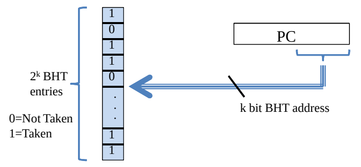
- Why does prediction work?
  - Underlying algorithm has regularities
  - Data that is being operated on has regularities
  - Instruction sequence has redundancies that are artifacts of way that humans/compilers think about problems
- Is dynamic branch prediction better than static branch prediction?
  - Seems to be (most modern processor use it)
  - There are a small number of important branches in programs which have dynamic behavior

81

Dynamic Branch Prediction

Simplest Dynamic Predictor:
Branch History Table: Lower bits of PC address index table of 1-bit values

Keeps track of whether or not branch taken last time
No address check

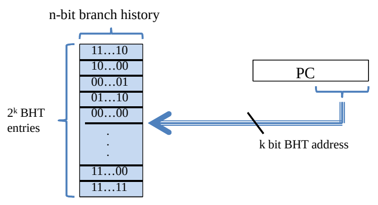


Problem: in a loop, 1-bit BHT will cause two mispredictions (average loop has only 9 iterations before exit):

End of loop case, when it exits instead of looping as before
First time through loop on *next* time through code, when it predicts exit instead of looping

82

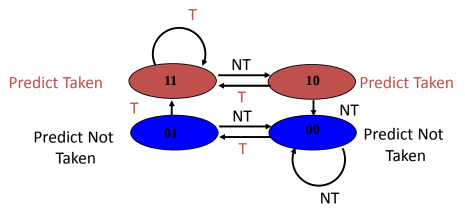
Multi-bit Branch History



In general, there is little performance improvement Beyond n=2

A two-bit branch predictor

- Change prediction only if get misprediction *twice*

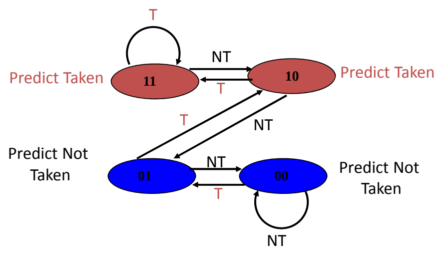


- Adds *hysteresis* to decision making process
- Many other two-bit prediction schemes are possible

84

Another two-bit branch predictor

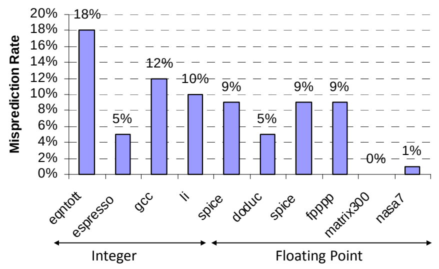
- Two-bit saturating counter (Smith Predictor)



85

2-bit BHT Table Predictor Accuracy

- Causes of Misprediction:
  - Wrong guess for that branch
  - Got branch history of wrong branch when index the table (aliasing, due to limited table size)
- 4096 entry table:



86

Correlated Branch Prediction

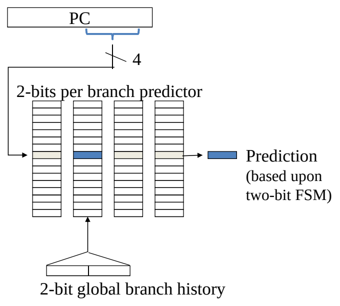
- Idea: track the outcome of the *m* most recently executed branches (globally), and use that pattern to select the proper *n*-bit branch history table
- In general, (*m*,*n*) predictor means use last *m (global)* branch outcomes to select between 2^m history tables, each with *n*-bit counters
  - Thus, old 2-bit BHT is a (0,2) predictor
- Global Branch History: *m*-bit shift register keeping T/NT status of last *m* branches.
- Each entry in table has *m* *n*-bit predictors (local branch history).

87

Example: A (2,2) Branch Predictor

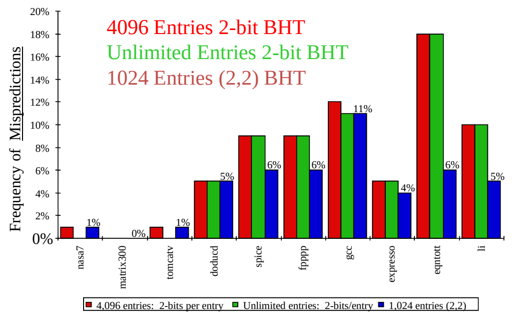
(2,2) predictor

- Behavior of recent branches selects between four predictions of next branch, updating just that prediction



88

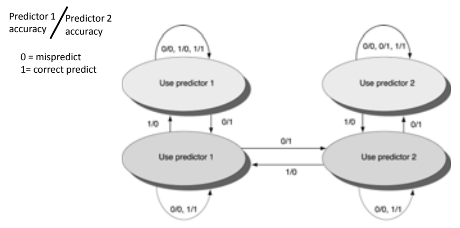
Branch Predictor Accuracy



89

Tournament Branch Predictor

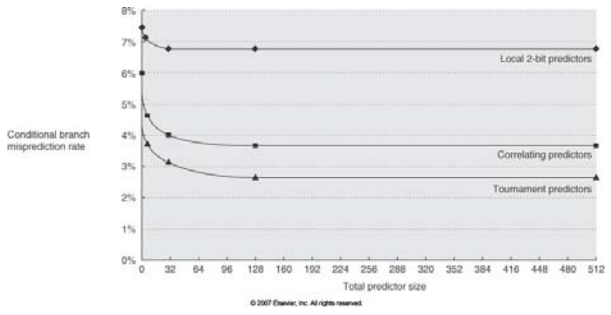
- Multilevel branch predictor
- Use *n*-bit saturating counter to choose between predictors
- Usual choice between global and local predictors



90

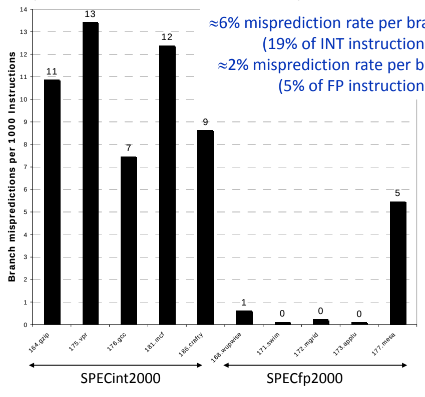
Branch Predictor Performance as a Function of Size (total # of Bits)

(SPEC89 Benchmarks)



91

Pentium 4 Misprediction Rate
(per 1000 instructions, not per branch)



≈6% misprediction rate per branch SPECint
(19% of INT instructions are branch)
≈2% misprediction rate per branch SPECfp
(5% of FP instructions are branch)

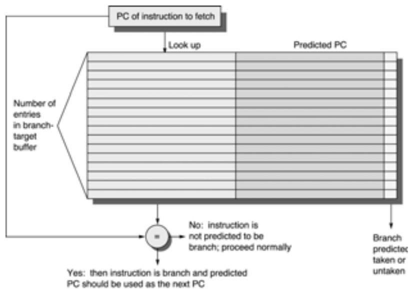
92

Branch Prediction—What about the Branch Target Address(BTA)?

- Branch Prediction is of no value unless we know the BTA
- Branch target calculation is costly and stalls the instruction fetch.
- A Branch Target Buffer (BTB) can store previously computed BTAs
- The BTA of a taken branch is stored in the BTB
- For subsequent executions of this branch, the BTA can be "looked up" in the BTB
- If the branch was predicted taken, instruction fetch continues at the predicted PC

93

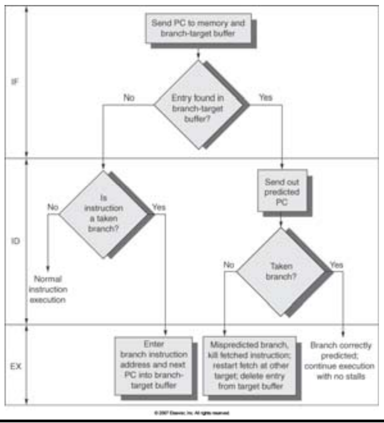
Branch Target Buffer (BTB)


PC of instruction to fetch
Look up
Predicted PC
Number of entries in branch-target buffer
=
No: instruction is not predicted to be branch; proceed normally
Yes: then instruction is branch and predicted PC should be used as the next PC
Branch predicted taken or untaken

Often, BTB is used in conjunction with Dynamic Prediction
- BTB provides fast prediction and BTA in fetch stage
- Dynamic Predictor provides more accurate prediction in decode stage

94

BTB Flowchart


IF
ID
EX
Send PC to memory and branch-target buffer
Entry found in branch-target buffer?
No
Yes
Is instruction a taken branch?
No
Yes
Normal instruction execution
Send out predicted PC
Taken branch?
No
Yes
Enter branch instruction address and next PC into branch-target buffer
Mispredicted branch, kill fetched instruction; restart fetch at other target; delete entry from target buffer
Branch correctly predicted; continue execution with no stalls
© 2007 Elsevier, Inc. All rights reserved.

Branch Prediction Summary

- Dynamic Prediction is essential in modern high-performance processors
- Branch History Table: 2 bits for loop accuracy
- Correlation: Recently executed branches correlated with next branch
- Tournament predictors take insight to next level, by using multiple predictors
  - usually one based on global information and one based on local information, and combining them with a selector
  - Tournament predictors using ≈ 30K bits are in processors like the Power5 and Pentium 4
- Branch Target Buffer: include branch address & prediction

96