Lecture 4: EITF20 Computer Architecture

Anders Ardö

EIT – Electrical and Information Technology, Lund University

November 6, 2013

Outline

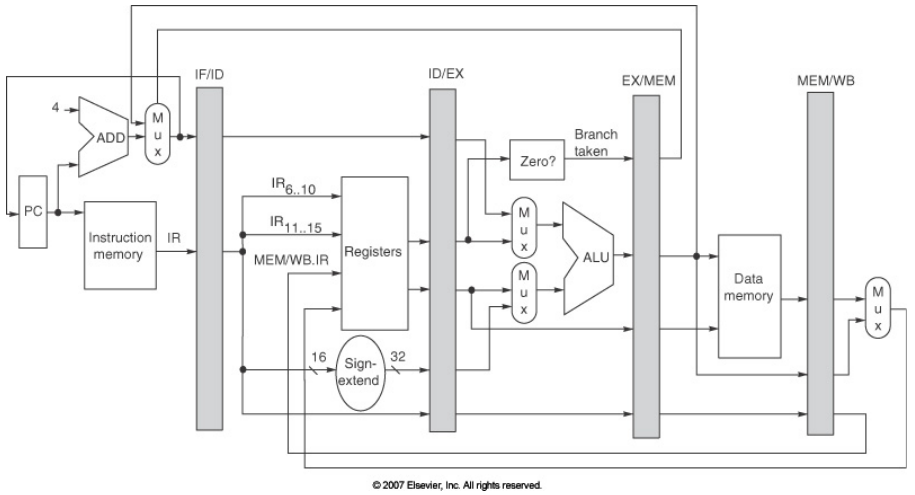
1. Reiteration
2. MIPS R4000
3. Instruction Level Parallelism
4. Branch Prediction
5. Dependencies
6. Instruction Scheduling
7. Scoreboard

Previous lecture

Introduction to pipeline basics

- General principles of pipelining
  (review from EIT070 Computer Organization)
- Techniques to avoid pipeline stalls due to hazards
- What makes pipelining hard to implement?
- Support of multi-cycle instructions in a pipeline

A Pipelined MIPS Datapath



Pipelining Lessons

- Pipelining doesn't help latency of a single instruction, it helps throughput of the entire workload
- Pipeline rate is limited by the slowest pipeline stage
- Multiple instructions are executing simultaneously
- Potential speedup = Number of pipe stages
  - Unbalanced lengths of pipe stages reduces speedup
  - Time to fill pipeline and time to drain reduces speedup
  - Hazards reduces speedup

Summary

Pipelining:

- Speeds up throughput, not latency
- Speedup $\leq$ #stages
- Hazards limit performance, generate stalls:
  - Structural: need more HW
  - Data (RAW,WAR,WAW): need forwarding and compiler scheduling
  - Control: delayed branch, branch prediction

$$Speedup = \frac{CPI_{unpipelined}}{\text{Ideal CPI} + \text{\# stall-cycles/instr}} * \frac{T_{C_{unpipelined}}}{T_{C_{pipelined}}}$$

$$\approx \frac{\#stages}{1 + \text{\# stall-cycles/instruction}}$$

Complications:

- Precise exceptions may be difficult to implement
- The instruction set can be more or less suited for pipelining

Questions!

QUESTIONS?

COMMENTS?

Lecture 4 agenda

Appendix A.6-A.7, and Chapters 2.1-2.6, 2.9 in "Computer Architecture"

1. Reiteration
2. MIPS R4000
3. Instruction Level Parallelism
4. Branch Prediction
5. Dependencies
6. Instruction Scheduling
7. Scoreboard

Lecture 4

- A case study of a medium deep pipeline: the MIPS R4000
- Introduction to Instruction Level Parallelism: ILP
- Branch prediction
- Instruction scheduling

Outline

1 Reiteration
2 MIPS R4000
3 Instruction Level Parallelism
4 Branch Prediction
5 Dependencies
6 Instruction Scheduling
7 Scoreboard

The MIPS R4000

- **R4000 - MIPS64**:
  - First (1992) true 64-bit architecture (addresses and data)
  - Clock frequency (1997): 100 MHz-250 MHz
  - Medium deep 8 stage pipeline (superpipelined)
- ***Implications because of the deeper pipeline***:
  - load latency: 2 cycles
    branch latency: 3 cycles (incl one delay slot)
    $\Longrightarrow$ High demands on the compiler
  - Bypassing (forwarding) from more stages
  - More instructions “in flight” in pipeline
- Faster clock, larger latencies, more stalls - Win or loose?
- ***Performance equation:*** **CPI $*$ T_C must be lower for the longer pipeline to make it worthwhile**

High Clock Frequency Implications

- 200 MHz clock frequency corresponds to 5 ns cycle time in each pipe stage.
- While an ALU can be clocked in this frequency, the memory accesses become main bottlenecks.

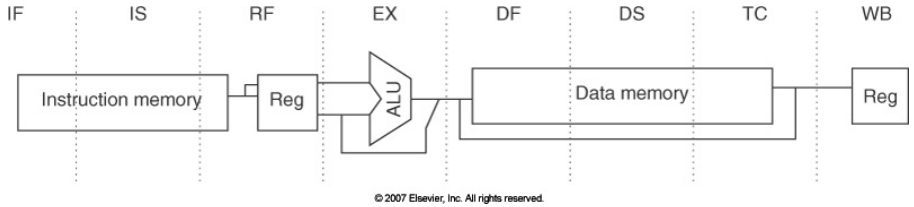
The MIPS R4000 has 8-16 Kbytes I- and D-caches
Extra pipeline stages comes from decomposing memory access

Operations at a cache access:

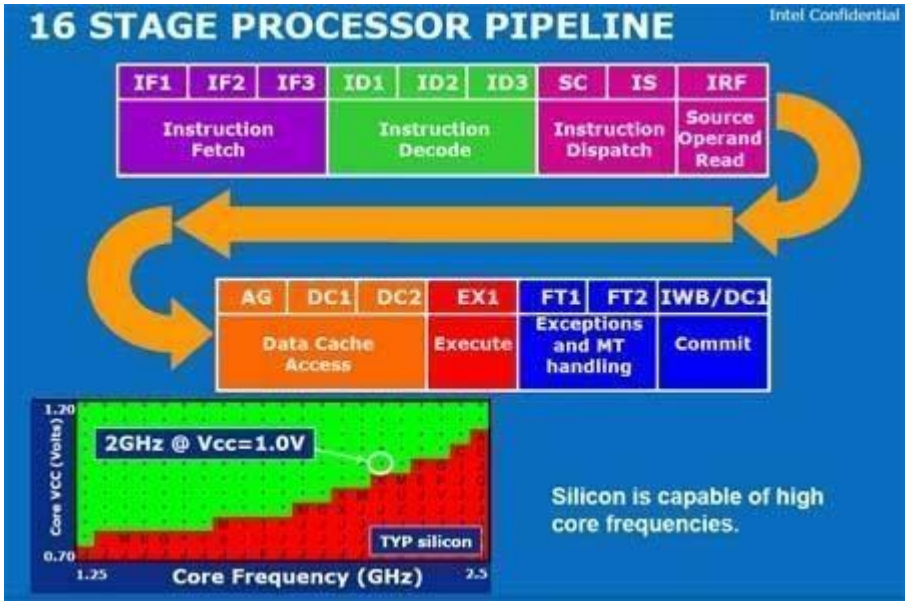
- Address translation
- Cache look-up
- Hit/miss detection

The MIPS R4000 Pipeline

- Instruction fetch is done in the IF and IS stages speculatively independent of a cache hit or miss
- RF performs the same as ID in the classic 5-stage pipeline but also check for instruction cache hit/miss
- The MEM stage is divided into three stages (DF, DS, and TC). Cache misses are detected in TC (Tag check)
- WB is the same as in classic 5-stage pipeline

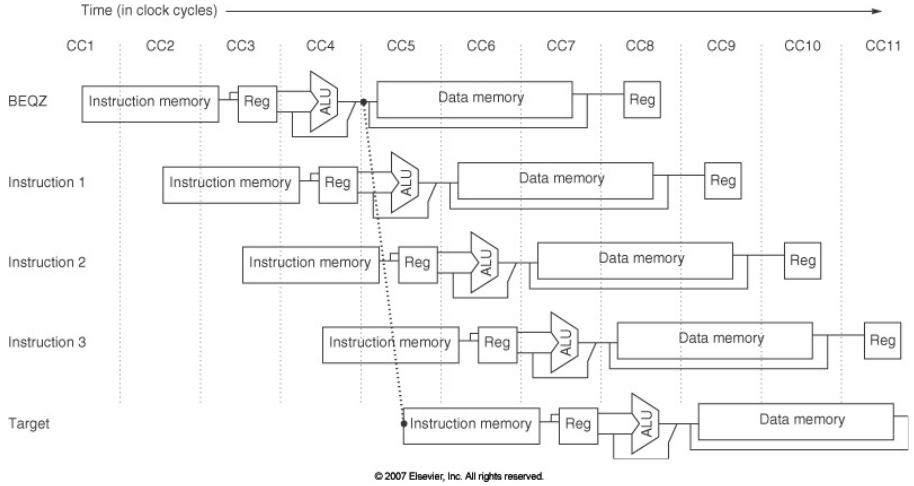


CPU chip architecture - Intel Atom



Branch Penalties

- *One* branch delay slot
- *Predict-Not taken scheme* squashes the next two ***sequentially fetched instructions*** if the branch is taken

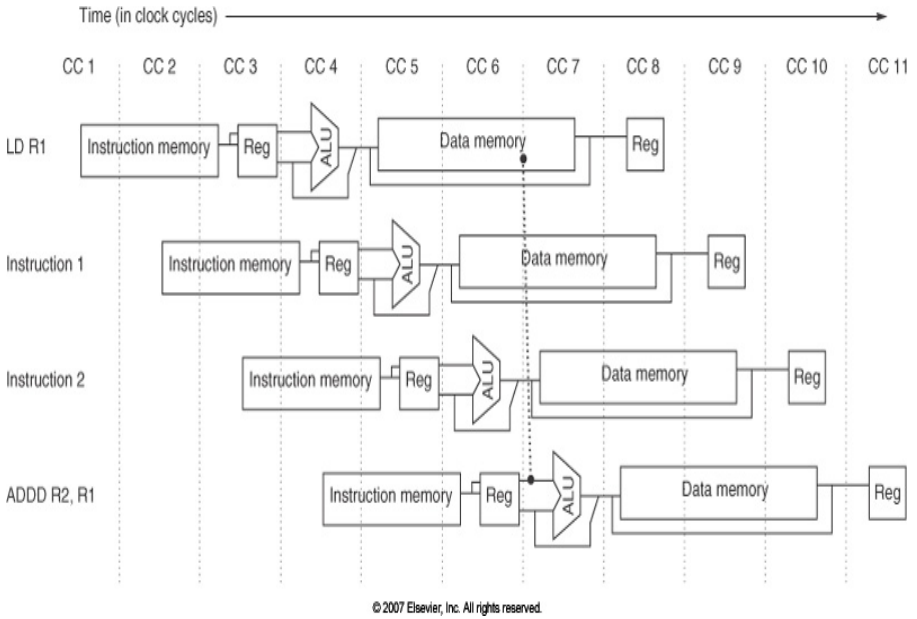


Branch Penalties

branch taken									
Instruction	Clock number								
	1	2	3	4	5	6	7	8	9
Branch instr	IF	IS	RF	EX	DF	DS	TC	WB	
Delay slot		IF	IS	RF	EX	DF	DS	TC	WB
Squash			IF	IS	s	s	s	s	s
Squash				IF	s	s	s	s	s
Branch target					IF	IS	RF	EX	DF

branch not taken									
Instruction	Clock number								
	1	2	3	4	5	6	7	8	9
Branch instr	IF	IS	RF	EX	DF	DS	TC	WB	
Delay slot		IF	IS	RF	EX	DF	DS	TC	WB
Branch instr +2			IF	IS	RF	EX	DF	DS	TC
Branch instr +3				IF	IS	RF	EX	DF	DS

Load Penalties

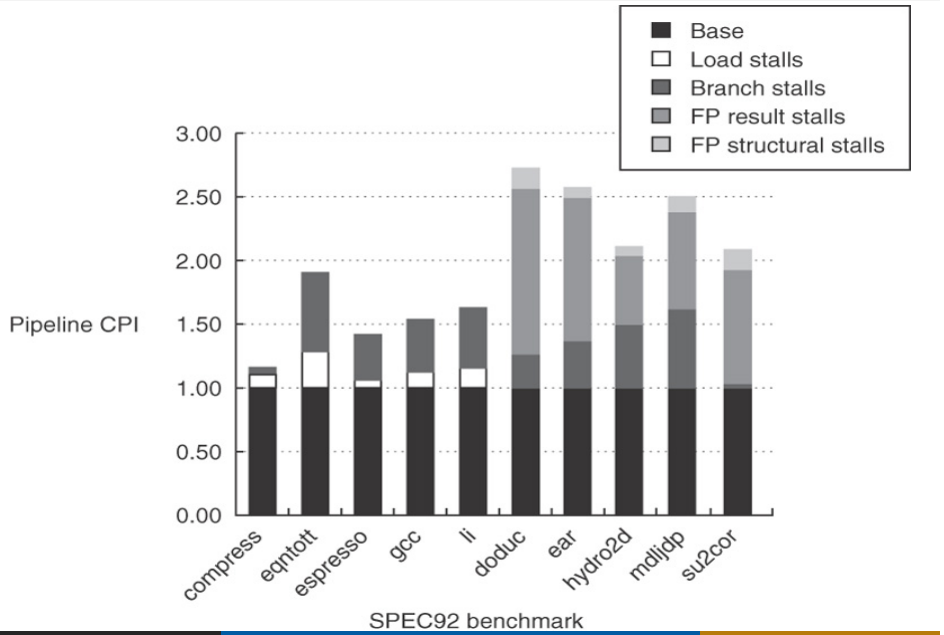


Load Penalties

Instruction	Clock number								
	1	2	3	4	5	6	7	8	9
LD R1,...	IF	IS	RF	EX	DF	DS	TC	WB	
DADD R2,R1,R4		IF	IS	RF	stall	stall	EX	DF	DS
DSUB R3,R1,R5			IF	IS	stall	stall	RF	EX	DF

- Data is available after the DS stage (second stage in 'Data memory')
- $\Longrightarrow$ 2 load delay slots
- Four bypasses instead of two because of the two extra pipe stages to read from memory.

R4000 Performance



R4000 Performance

	CPI penalties			
Average CPI	Load	Branch	FP data hazard	FP struct hazard
2.00	0.10	0.36	0.46	0.09

(SPEC92 CPI measurements)

- The penalty for control hazards is very high for integer programs (up to 0.61)
- The penalty for FP data hazards is also high
- The higher clock frequency compensates for the higher CPI

Outline

1. Reiteration
2. MIPS R4000
3. Instruction Level Parallelism
4. Branch Prediction
5. Dependencies
6. Instruction Scheduling
7. Scoreboard

Instruction Level Parallelism - ILP

ILP: Overlap execution of unrelated instructions: **Pipelining**
Two main approaches:

- DYNAMIC $\Longrightarrow$ hardware detects parallelism
- STATIC $\Longrightarrow$ software detects parallelism

Often a mix between both.

Pipeline CPI = Ideal CPI + Structural stalls
+ Data hazard stalls + Control stalls

Instruction Level Parallelism - ILP

Example:

```
DIVD   F0,F2,F4
ADDD   F10,F1,F2
SUBD   F8,F8,F14
```

- These instructions are independent and could be executed in parallel
- gcc has 17% control transfer instructions:
  - 5 sequential instructions / branch
  - We must look beyond a single basic block to get more ILP
- Loop level parallelism one opportunity, SW & HW

Loop-level Parallelism

```
for (i=1; i≤1000; i=i+1)
        x[i] = x[i] + 10;
```

- There is very little available parallelism *within* an iteration
- However, there is parallelism *between* loop iterations; each iteration is independent of the other

Potential speedup = 1000!!!

MIPS-code for the Loop

```
loop:  LD     F0, 0(R1)     ; F0 = array element
       ADDD   F4, F0, F2    ; Add scalar constant
       SD     F4, 0(R1)     ; Save result
       DADDUI R1, R1, #-8   ; decrement array ptr.
       BNE    R1,R2, loop   ; reiterate if R1 != R2
```

Instruction producing result	Instruction using result	Latency
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0
Integer op	Integer op	0
Integer op	Cond. branch	1

Loop Showing Stalls

```
1   loop:  LD     F0, 0(R1)     ; F0 = array element
2          stall
3          ADDD   F4, F0, F2    ; Add scalar constant
4          stall
5          stall
6          SD     F4, 0(R1)     ; Save result
7          DADDUI R1, R1, #-8   ; decrement array ptr.
8          stall
9          BNE    R1,R2 loop    ; reiterate if R1 != R2
10         nop
```

How can we get rid of the stalls?

Restructured Loop Minimizing Stalls

```
1   loop:  LD     F0, 0(R1)     ; F0 = array element
2          DADDUI R1, R1, #-8   ; decrement array ptr.
3          ADDD   F4, F0, F2    ; Add scalar constant
4          stall
5          BNE    R1, R2, loop  ; reiterate if R1 != R2
6          SD     F4, 8(R1)     ; Save result
```

- Swap DADDUI and SD by changing offset in SD
- 6 clock cycles per iteration
- Sophisticated compiler analysis required
- Can we do better?

Unroll Loop Four Times

```
1   loop:  LD     F0, 0(R1)
2          ADDD   F4, F0, F2
3          SD     F4, 0(R1)     ; drop DADDUI & BNE
4          LD     F6, -8(R1)
5          ADDD   F8, F6, F2
6          SD     F8, -8(R1)    ; drop DADDUI & BNE
7          LD     F10, -16(R1)
8          ADDD   F12, F10, F2
9          SD     F12, -16(R1)  ; drop DADDUI & BNE
10         LD     F14, -24(R1)
11         ADDD   F16, F14, F2
12         SD     F16, -24(R1)
13         DADDUI R1, R1, #-32  ; alter to 4*8
14         BNE    R1, R2, loop
15         NOP
```

- 15 + 4*(1+2) + 1 = 28 clock cycles, or 7 per iteration

Scheduled Unrolled Loop

```
1   loop:  LD      F0, 0(R1)
2          LD      F6, -8(R1)
3          LD      F10, -16(R1)
4          LD      F14, -24(R1)
5          ADDD    F4, F0, F2
6          ADDD    F8, F6, F2
7          ADDD    F12, F10, F2
8          ADDD    F16, F14, F2
9          SD      F4, 0(R1)
10         SD      F8, -8(R1)
11         DADDUI  R1, R1, #-32
12         SD      F12, 16(R1)    ; 16-32 = -16
13         BNE     R1, R2, loop
14         SD      F16, 8(R1)     ; 8-32 = -24
```

- 14 clock cycles, or 3.5 per iteration
- When is it safe to move instructions?

Why loop unrolling works

- Longer sequences of straight code without branches (longer basic blocks) allows for ***easier compiler static rescheduling***
- Longer basic blocks also facilitates dynamic rescheduling such as Scoreboard and Tomasulo's algorithm
- In practice a variant of software pipelining - modulo scheduling is used.

Outline

1 Reiteration
2 MIPS R4000
3 Instruction Level Parallelism
4 Branch Prediction
5 Dependencies
6 Instruction Scheduling
7 Scoreboard

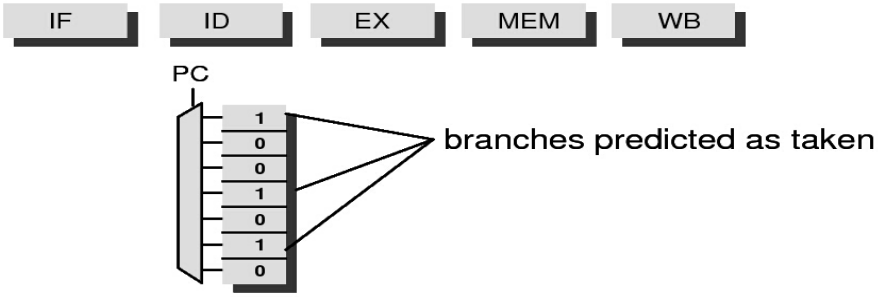
Dynamic Branch Prediction

- Branches limit performance because:
  - Branch penalties
  - Limit to available Instruction Level Parallelism
- Solution: ***Dynamic branch prediction*** to predict the outcome of conditional branches.
  Benefits:
  - Reduce the time to when the branch condition is known
  - Reduce the time to calculate the branch target address

Branch History Table

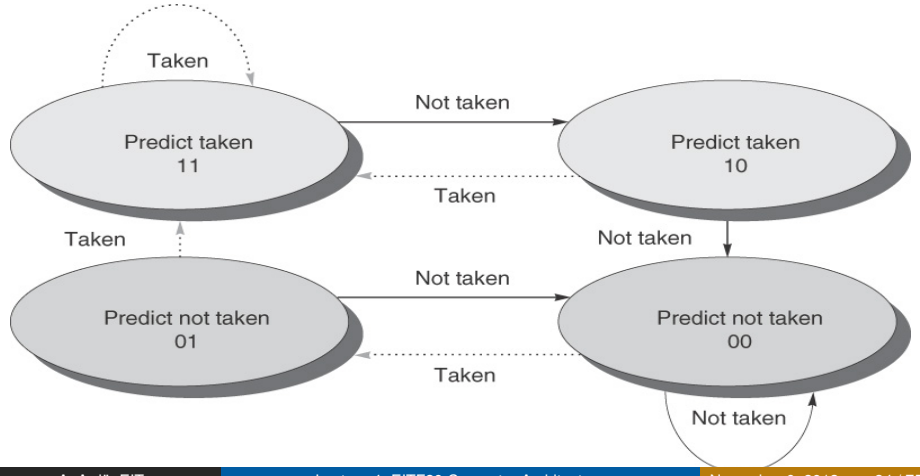
Simplest branch prediction

- The branch-prediction buffer is indexed by low order part of branch-instruction address
- The bit corresponding to a branch indicates whether the branch is predicted as taken or not
- When prediction is wrong: *invert bit*



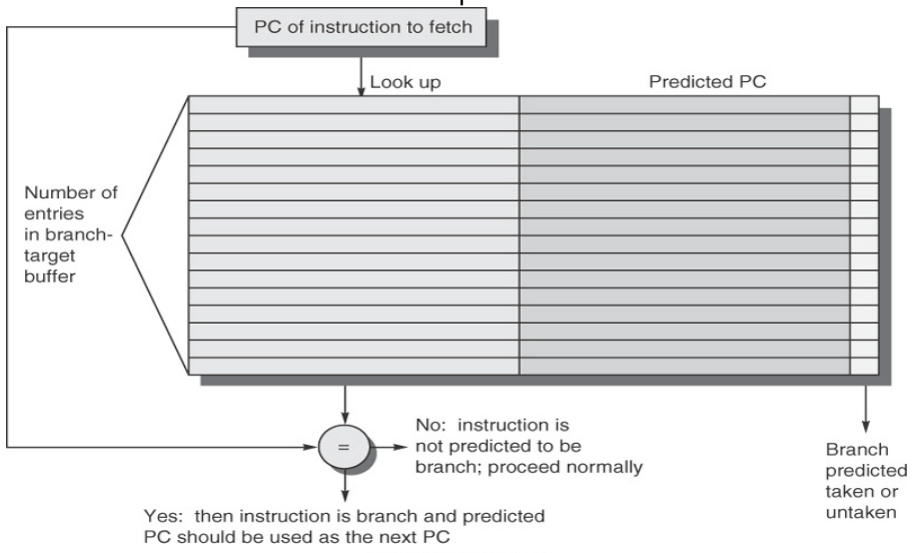
A two-bit prediction scheme

- Requires prediction to miss twice in order to change prediction $\Longrightarrow$ better performance
- 1%-18% miss prediction frequency for SPEC89
- Integer programs have higher miss frequency than FP programs

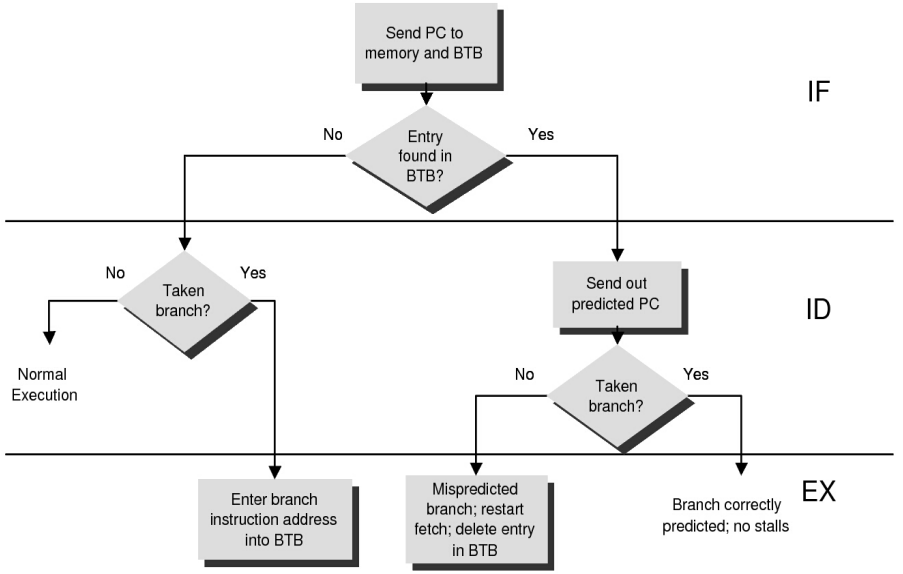


Branch Target Buffer - BTB

Predicts *branch target address* in the IF stage
Can be combined with 2-bit branch prediction



Branch Target Buffer (BTB) Algorithm



Branch prediction penalties

for a Branch Target Buffer scheme

Instruction in buffer	Prediction	Actual branch	Penalty cycles
Yes	Taken	Taken	0
Yes	Taken	Not taken	2
No		Taken	2
No		Not taken	0

Outline

1 Reiteration
2 MIPS R4000
3 Instruction Level Parallelism
4 Branch Prediction
5 Dependencies
6 Instruction Scheduling
7 Scoreboard

Dependencies

- Two instructions must be ***independent*** in order to execute in parallel
- There are three general types of dependencies that limit parallelism:
  - Data dependencies
  - Name dependencies
  - Control dependencies
- Dependencies are properties of the ***program***
- Whether a dependency leads to a hazard or not is a property of the ***pipeline implementation***

Data Dependencies

An instruction *j* is data dependent on instruction *i* if:

- Instruction *i* produces a result used by instr. *j*, or
- Instruction *j* is data dependent on instruction *k* and instr. *k* is data dependent on instr. *i*

Example:

```
LD    F0,0(R1)
ADDD  F4,F0,F2
SD    0(R1),F4
```

- Easy to detect for registers

Name Dependencies

Two instructions use same name (register or memory address) but do not exchange data

- Anti-dependence (WAR if hazard in HW)

```
ADDD    F2,F0,F2    ; Must execute before LD
LD      F0,0(R1)
```

- Output dependence (WAW if hazard in HW)

```
ADDD    F0,F2,F2    ; Must execute before LD
LD      F0,0(R1)
```


Control Dependencies

Determines order between an instruction and a branch instruction

Example:

if Test1 then { S1 }
if Test2 then { S2 }

S1 is control dependent on Test1
S2 is control dependent on Test2; but *not* on Test1

- We cannot move an instruction that is dependent on a branch *before* the branch instruction
- We cannot move an instruction not control dependent on a branch *after* the branch instruction

Outline

1 Reiteration
2 MIPS R4000
3 Instruction Level Parallelism
4 Branch Prediction
5 Dependencies
6 Instruction Scheduling
7 Scoreboard

Instruction scheduling

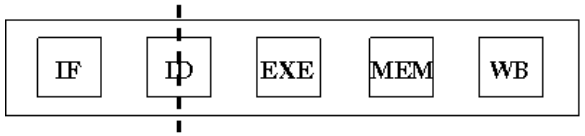
Scheduling is the process that determines when to start a particular instruction, when to read its operands, and when to write its result,
Target of scheduling: rearrange instructions to reduce stalls when data or control dependencies are present

- ***Static (compiler at compile-time) scheduling***:
  - Pro: May look into future; no HW support needed
  - Con: Cannot detect all dependencies, esp. across branches
- ***Dynamic (hardware at run-time) scheduling***:
  - Pro: Can potentially detect all dependencies
  - Con: Cannot look into the future; HW support needed which complicates the pipeline hardware

Dynamic instruction scheduling

- Key idea: allow instructions behind stall to proceed

```
DIVD    F0,F2,F4    ; takes long time
ADDD    F10,F0,F8   ; stalls waiting for F0
```



MULTD F12,F8,F14 ; Let this instr. bypass the ADDD

- MULTD is not data dependent on anything in the pipeline
- Enables out-of-order execution $\Longrightarrow$ out-of-order completion
- ID stage checks for structural and data dependencies
  - Scoreboard (CDC 6600 in 1963)
  - Tomasulo (IBM 360/91 in 1967)

Dynamic instruction scheduling

Why in hardware at run time?

- It works when we cannot know real dependence at compile time
- It makes the compiler simpler
- Code for one implementation runs well on another

Outline

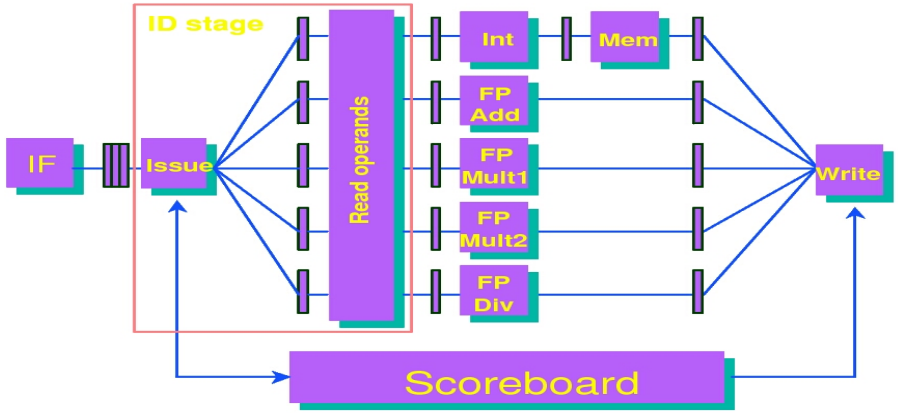
1 Reiteration
2 MIPS R4000
3 Instruction Level Parallelism
4 Branch Prediction
5 Dependencies
6 Instruction Scheduling
7 Scoreboard

Scoreboard pipeline

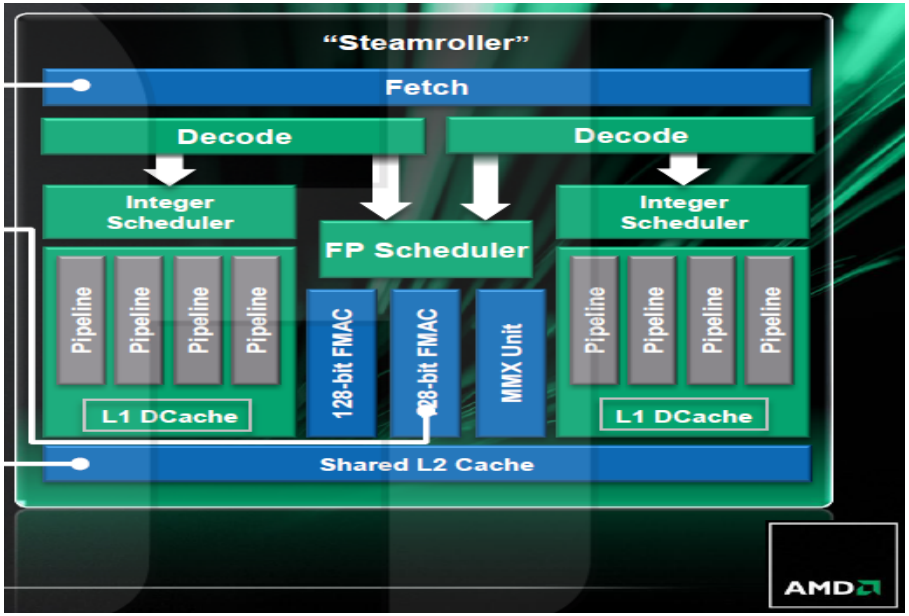
- Goal of scoreboarding is to maintain an execution rate of one instruction per clock cycle by executing an instruction as early as possible.
- Instructions execute out-of-order when there are sufficient resources and no data dependencies.
- A scoreboard is a hardware unit that keeps track of
  - the instructions that are in the process of being executed,
  - the functional units that are doing the executing,
  - and the registers that will hold the results of those units.
- A scoreboard centrally performs all hazard detection and resolution and thus controls the instruction progression from one step to the next.

Scoreboard pipeline

- **Issue**: Decode and check for structural hazards
- **Read operands**: wait until no data hazards, then read operands
- All data hazards are handled by the scoreboard



Modern CPU pipeline



Scoreboard complications

Out-of-order execution $\Longrightarrow$ WAR & WAW hazards

- Solutions for WAR:
  - Stall instruction in the Write stage until all previously issued instructions (with a WAR hazard) have read their operands
- For WAW, must detect hazard: stall in Issue until other instruction completes
- RAW hazards handled in Read Operands
- Scoreboard keeps track of dependencies and state of operations

Scoreboard functionality

- **Issue**: An instruction is issued if:
  - The needed functional unit is free (there is no structural hazard)
  - No functional unit has a destination operand equal to the destination of the instruction (resolves WAW hazards)
- **Read**: Wait until no data hazards, then read operands
  - Performed in parallel for all functional units
  - Resolves RAW hazards dynamically
- **EX**: normal execution
  - Notify the scoreboard when ready
- **Write**: The instruction can update destination if:
  - All earlier instructions have read their operands (resolves WAR hazards)

Data structures in the scoreboard

- 1. ***Instruction status*** – keeps track of which of the 4 steps the instruction is in
- 2. ***Functional unit status*** – Indicates the state of the functional unit (FU). 9 fields for each FU:
  - *Busy*: Indicates whether the unit is busy or not
  - *Op*: Operation to perform in the unit (e.g. add or sub)
  - *Fi*: Destination register name
  - *Fj, Fk*: Source register names
  - *Qj, Qk*: Name of functional unit producing regs Fj, Fk
  - *Rj, Rk*: Flags indicating when Fj and Fk are ready
- 3. ***Register result status*** – Indicates which functional unit will write each register, if any.

Scoreboard Example

Instruction status

Instruction	j	k	Issue	Read ops	Exec. compl.	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				

Functional unit status

Time	Name	Busy	Op	dest Fi	src 1 Fj	src 2 Fk	FUsrc1 Qj	FUsrc2 Qk	Fj? Rj	Fk? Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status

	F0	F2	F4	F6	F8	F10	...	F30
FU								

Clock: 0

Detailed Scoreboard Pipeline Control

See figure A.54 in “Computer Architecture”

Instruction status	Wait until	Bookkeeping
Issue	Not Busy [FU] and not Result [D]	Busy[FU]←yes; Op[FU]←op; Fi[FU]←D; Fj[FU]←S1; Fk[FU]←S2; Qj←Result[S1]; Qk←Result[S2]; Rj←not Qj; Rk←not Qk; Result[D]←FU;
Read operands	Rj and Rk	Rj←No; Rk←No; Qj←0; Qk←0
Execution complete	Functional unit done	
Write result	∀f((Fj[f] ≠ Fi[FU] or Rj[f] = No) & (Fk[f] ≠ Fi[FU] or Rk[f] = No))	∀f(if Qj[f]=FU then Rj[f]←Yes); ∀f(if Qk[f]=FU then Rk[f]←Yes); Result[Fi[FU]]←0; Busy[FU]←No

Scoreboard Example, CP 1

Instruction status

Instruction	j	k	Issue	Read ops	Exec. compl.	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				

Functional unit status

Time	Name	Busy	Op	dest Fi	src 1 Fj	src 2 Fk	FUsrc1 Qj	FUsrc2 Qk	Fj? Rj	Fk? Rk
	Integer	Yes	Load	F6		R2				Yes
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status

	F0	F2	F4	F6	F8	F10	...	F30
FU				Integer				

Clock: 1

Scoreboard Example, CP 2

Issue 2nd load?

Instruction status

Instruction		j	k	Issue	Read ops	Exec. compl.	Write result
LD	F6	34+	R2	1	2		
LD	F2	45+	R3				
MULTD	F0	F2	F4				
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status

Time	Name	Busy	Op	dest Fi	src 1 Fj	src 2 Fk	FUsrc1 Qj	FUsrc2 Qk	Fj? Rj	Fk? Rk
	Integer	Yes	Load	F6		R2				No
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status

	F0	F2	F4	F6	F8	F10	...	F30
FU				Integer				

Clock: 2

Scoreboard Example, CP 3

Issue MULT?

Instruction status

Instruction		j	k	Issue	Read ops	Exec. compl.	Write result
LD	F6	34+	R2	1	2	3	
LD	F2	45+	R3				
MULTD	F0	F2	F4				
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status

Time	Name	Busy	Op	dest Fi	src 1 Fj	src 2 Fk	FUsrc1 Qj	FUsrc2 Qk	Fj? Rj	Fk? Rk
	Integer	Yes	Load	F6		R2				No
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status

	F0	F2	F4	F6	F8	F10	...	F30
FU				Integer				

Clock: 3

Scoreboard Example, CP 4

Instruction status

Instruction		j	k	Issue	Read ops	Exec. compl.	Write result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3				
MULTD	F0	F2	F4				
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status

Time	Name	Busy	Op	dest Fi	src 1 Fj	src 2 Fk	FUsrc1 Qj	FUsrc2 Qk	Fj? Rj	Fk? Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status

	F0	F2	F4	F6	F8	F10	...	F30
FU				-				

Clock: 4

Scoreboard Example, CP 5

Instruction status

Instruction		j	k	Issue	Read ops	Exec. compl.	Write result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5			
MULTD	F0	F2	F4				
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status

Time	Name	Busy	Op	dest Fi	src 1 Fj	src 2 Fk	FUsrc1 Qj	FUsrc2 Qk	Fj? Rj	Fk? Rk
	Integer	Yes	Load	F2		R3				Yes
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status

	F0	F2	F4	F6	F8	F10	...	F30
FU		Integer						

Clock: 5

Scoreboard Example, CP 6

Instruction status

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

Functional unit status

Time	Name	Busy	Op	dest Fi	src 1 Fj	src 2 Fk	FUsrc1 Qj	FUsrc2 Qk	Fj? Rj	Fk? Rk
	Integer	Yes	Load	F2		R3				No
	Mult1	Yes	Mult	F0	F2	F4	Integer		No	Yes
	Mult2	No								
	Add	No								
	Divide	No								

Register result status

	F0	F2	F4	F6	F8	F10	...	F30
FU	Mult1	Integer						

Clock: 6

Scoreboard Example, CP 7

Instruction status

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

Functional unit status

Time	Name	Busy	Op	dest Fi	src 1 Fj	src 2 Fk	FUsrc1 Qj	FUsrc2 Qk	Fj? Rj	Fk? Rk
	Integer	Yes	Load	F2		R3				No
	Mult1	Yes	Mult	F0	F2	F4	Integer		No	Yes
	Mult2	No								
	Add	Yes	Sub	F8	F6	F2		Integer	Yes	No
	Divide	No								

Register result status

	F0	F2	F4	F6	F8	F10	...	F30
FU	Mult1	Integer			Add			

Clock: 7

Scoreboard Example, CP 8

Instruction status

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

Functional unit status

Time	Name	Busy	Op	dest Fi	src 1 Fj	src 2 Fk	FUsrc1 Qj	FUsrc2 Qk	Fj? Rj	Fk? Rk
	Integer	No								
	Mult1	Yes	Mult	F0	F2	F4	-		Yes	Yes
	Mult2	No								
	Add	Yes	Sub	F8	F6	F2		-	Yes	Yes
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status

	F0	F2	F4	F6	F8	F10	...	F30
FU	Mult1	-			Add	Divide		

Clock: 8

Scoreboard Example, CP 9

Instruction status

Instruction		j	k	Issue	Read ops	Exec. compl.	Write result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9		
SUBD	F8	F6	F2	7	9		
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2				

Read operands for MULT & SUB

Issue ADDD?

Functional unit status

Time	Name	Busy	Op	dest Fi	src 1 Fj	src 2 Fk	FUsrc1 Qj	FUsrc2 Qk	Fj? Rj	Fk? Rk
	Integer	No								
10	Mult1	Yes	Mult	F0	F2	F4			No	No
	Mult2	No								
2	Add	Yes	Sub	F8	F6	F2			No	No
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status

	F0	F2	F4	F6	F8	F10	...	F30
FU	Mult1				Add	Divide		

Clock: 9

Scoreboard Example, CP 11

Instruction status

Instruction		j	k	Issue	Read ops	Exec. compl.	Write result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9		
SUBD	F8	F6	F2	7	9	11	
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2				

SUBD completes execution

Functional unit status

Time	Name	Busy	Op	dest Fi	src 1 Fj	src 2 Fk	FUsrc1 Qj	FUsrc2 Qk	Fj? Rj	Fk? Rk
	Integer	No								
8	Mult1	Yes	Mult	F0	F2	F4			No	No
	Mult2	No								
0	Add	Yes	Sub	F8	F6	F2			No	No
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status

	F0	F2	F4	F6	F8	F10	...	F30
FU	Mult1				Add	Divide		

Clock: 11

Scoreboard Example, CP 12

Instruction status

Instruction		j	k	Issue	Read ops	Exec. compl.	Write result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9		
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2				

Read operands for DIVD?

Functional unit status

Time	Name	Busy	Op	dest Fi	src 1 Fj	src 2 Fk	FUsrc1 Qj	FUsrc2 Qk	Fj? Rj	Fk? Rk
	Integer	No								
7	Mult1	Yes	Mult	F0	F2	F4			No	No
	Mult2	No								
	Add	No							-	-
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status

	F0	F2	F4	F6	F8	F10	...	F30
FU	Mult1				-	Divide		

Clock: 12

Scoreboard Example, CP 13

Instruction status

Instruction		j	k	Issue	Read ops	Exec. compl.	Write result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9		
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2	13			

Issue ADDD

Functional unit status

Time	Name	Busy	Op	dest Fi	src 1 Fj	src 2 Fk	FUsrc1 Qj	FUsrc2 Qk	Fj? Rj	Fk? Rk
	Integer	No								
6	Mult1	Yes	Mult	F0	F2	F4			No	No
	Mult2	No								
	Add	Yes	Add	F6	F8	F2			Yes	Yes
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status

	F0	F2	F4	F6	F8	F10	...	F30
FU	Mult1			Add		Divide		

Clock: 13

Scoreboard Example, CP 16

Instruction status

Instruction		j	k	Issue	Read ops	Exec. compl.	Write result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9		
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2	13	14	16	

Can ADDD write result?

Functional unit status

Time	Name	Busy	Op	dest Fi	src 1 Fj	src 2 Fk	FUsrc1 Qj	FUsrc2 Qk	Fj? Rj	Fk? Rk
	Integer	No								
3	Mult1	Yes	Mult	F0	F2	F4			No	No
	Mult2	No								
0	Add	Yes	Add	F6	F8	F2			No	No
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status

	F0	F2	F4	F6	F8	F10	...	F30
FU	Mult1			Add		Divide		

Clock: 16

Scoreboard Example, CP 17

Instruction status

Instruction		j	k	Issue	Read ops	Exec. compl.	Write result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9		
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2	13	14	16	

ADDD stalls, waiting for DIVD to read F6

Resolves a WAR hazard!

Functional unit status

Time	Name	Busy	Op	dest Fi	src 1 Fj	src 2 Fk	FUsrc1 Qj	FUsrc2 Qk	Fj? Rj	Fk? Rk
	Integer	No								
2	Mult1	Yes	Mult	F0	F2	F4			No	No
	Mult2	No								
	Add	Yes	Add	F6	F8	F2			No	No
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status

	F0	F2	F4	F6	F8	F10	...	F30
FU	Mult1			Add		Divide		

Clock: 17

Scoreboard Example, CP 20

Instruction status

Instruction		j	k	Issue	Read ops	Exec. compl.	Write result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9	19	20
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2	13	14	16	

Functional unit status

Time	Name	Busy	Op	dest Fi	src 1 Fj	src 2 Fk	FUsrc1 Qj	FUsrc2 Qk	Fj? Rj	Fk? Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	Yes	Add	F6	F8	F2			No	No
	Divide	Yes	Div	F10	F0	F6	-		Yes	Yes

Register result status

	F0	F2	F4	F6	F8	F10	...	F30
FU	-			Add		Divide		

Clock: 20

Scoreboard Example, CP 21

Instruction status

Instruction		j	k	Issue	Read ops	Exec. compl.	Write result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9	19	20
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8	21		
ADDD	F6	F8	F2	13	14	16	

Functional unit status

Time	Name	Busy	Op	dest Fi	src 1 Fj	src 2 Fk	FUsrc1 Qj	FUsrc2 Qk	Fj? Rj	Fk? Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	Yes	Add	F6	F8	F2			No	No
	Divide	Yes	Div	F10	F0	F6			No	No

Register result status

	F0	F2	F4	F6	F8	F10	...	F30
FU				Add		Divide		

Clock: 21

Scoreboard Example, CP 22

Instruction status

Instruction		j	k	Issue	Read ops	Exec. compl.	Write result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9	19	20
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8	21		
ADDD	F6	F8	F2	13	14	16	22

Now ADDD can safely write its result in F6

Functional unit status

Time	Name	Busy	Op	dest Fi	src 1 Fj	src 2 Fk	FUsrc1 Qj	FUsrc2 Qk	Fj? Rj	Fk? Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
40	Divide	Yes	Div	F10	F0	F6			No	No

Register result status

	F0	F2	F4	F6	F8	F10	...	F30
FU				-		Divide		

Clock: 22

Scoreboard Example, CP 62

Instruction status

Instruction		j	k	Issue	Read ops	Exec. compl.	Write result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9	19	20
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8	21	61	62
ADDD	F6	F8	F2	13	14	16	22

Functional unit status

Time	Name	Busy	Op	dest Fi	src 1 Fj	src 2 Fk	FUsrc1 Qj	FUsrc2 Qk	Fj? Rj	Fk? Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status

	F0	F2	F4	F6	F8	F10	...	F30
FU						-		

Clock: 62

Factors that limit performance

The scoreboard technique is limited by:

- The amount of parallelism available in code
- The number of scoreboard entries (*window size*)
  - A large window can look ahead across more instructions
- The number and types of functional units
  - Contention for functional units leads to structural hazards
- The presence of anti-dependencies and output dependencies
  - These lead to WAR and WAW hazards that are handled by stalling the instruction in the Scoreboard
- Number of datapaths to registers

Scoreboard Summary

Main advantage:

- managing multiple FUs
- out-of-order execution of multi-cycle operations
- maintaining all data dependencies (RAW, WAW, WAR)

Scoreboard limitations:

- single issue scheme, however: scheme is extendable to multiple-issue
- in-order issue
- anti-dependencies and output dependencies may lead to WAR and WAW stalls,
- no forwarding hardware $\Longrightarrow$ all results go through the registers