Advanced Pipelining and Instruction-Level Paralelism

Riferimenti bibliografici

" "Computer architecture, a quantitative approach", Hennessy & Patterson: (Morgan Kaufmann eds.)

Introduction

- Pipelining become universal technique in 1985
  - Overlaps execution of instructions
  - Exploits "Instruction Level Parallelism"
- Beyond this, there are two main approaches:
  - Hardware-based dynamic approaches
    - Used in server and desktop processors
  - Compiler-based static approaches
    - More used in embedded market (but IA-64 architecture and Intel's Itanium use these approaches)

Instruction-Level Parallelism

- When exploiting instruction-level parallelism, goal is to maximize CPI
  - Pipeline CPI =
    - Ideal pipeline CPI +
    - Structural stalls +
    - Data hazard stalls +
    - Control stalls

Instruction-Level Parallelism

- Parallelism with basic block - a straight-line code sequence with no branches in except to the entry and no branches out except at the exit - is limited
  - Typical size of basic block = 3-7 instructions
  - Must optimize across multiple basic block

```
for(i=0;i<=1000;i=i+1)
        x[i]=x[i]+y[i]
```

```
; r1 = address of the last element of x
; r2 = address of the last element of y
Loop:
  ld   f0,0(r1) ; f0=array element x[i]
  ld   f2, 0(r2) ; f2=array element y[i]
  addd  f4,f0,f2 ; add scalar in f2
  sd    0(r1),f4 ; store result
  subi  r1,r1,8  ; descr. Pointer
  subi  r2,r2,8  ; descr. Pointer
  bnez  r1, Loop ; branch r1!=0
```

basic block

Instruction-Level Parallelism

```
for(i=0;i<=1000;i=i+1)
      x[i]=x[i]+y[i]
```

- Loop-Level Parallelism
  - Unroll loop statically or dynamically
  - Use SIMD (vector processors and GPUs)

Data Dependence

- Challenges:
  - Data dependency
    - Instruction *j* is data dependent on instruction *i* if
      - Instruction *i* produces a result that may be used by instruction *j*
      - Instruction *j* is data dependent on instruction *k* and instruction *k* is data dependent on instruction *i*
- Dependent instructions cannot be executed simultaneously

```
; r1 = address of the last element of x
; f2 = s
Loop:
  ld    f0,0(r1) ; f0=array element
  addd  f4,f0,f2 ; add scalar in f2
  sd    0(r1),f4 ; store result
  subi  r1,r1,8  ; descr. Pointer
  bnez  r1, Loop ; branch r1!=0
```

ld produces f0 that is used by addd
addd produces f4 that is used by sd

subi produces r1 that is used by bnez

Dependences

- Determining dependences among instructions is critical to defining the amount of parallelism existing in a program.
- If two instructions are dependent, they cannot execute in parallel: they must be executed in order or only partially overlapped.
- Three different types of dependences:
  - Data Dependences (or True Data Dependences)
  - Name Dependences
  - Control Dependences

Name dependences

- Name dependence occurs when 2 instructions use the same register or memory location (called **name**), but there is no flow of data between the instructions associated with that name.
- Two types of name dependences between an instruction **i** that precedes instruction **j** in program order:
  - **Antidependence:** when **j** writes a register or memory location that instruction **i** reads (WAR). The original instructions ordering must be preserved to ensure that **i** reads the correct value.

Example:

```
i:    dadd r3, r2, r4
j:    ld r2, X(r6)
```

Name dependences

- **Output Dependence:** when **i** and **j** write the same register or memory location (WAW). The original instructions ordering must be preserved to ensure that the value finally written corresponds to **j.**

Example:

```
i:    dadd r3, r2, r4
j:    ld r3, X(r6)
```

Name dependences

- Name dependences are not true data dependences, since there is no value (no data flow) being transmitted between instructions.
- If the name (register number or memory location) used in the instructions could be changed, the instructions do not conflict.

Example:

```
i:    dadd r3, r2, r4          i:    dadd r3, r2, r4
j:    ld r2, X(r6)      →      j:    ld r7, X(r6)
j+1   dadd r9, r2, r8          j+1   dadd r9,r7,r8
```

Name dependences

- Dependences trough memory locations are more difficult to detect (**"memory disambiguation"** problem), since two addresses may refer to the same location but can look different.

  Example:

```
                daddi r6, r0, 1008
                daddi r7, r0, 1000
                      ....
i:              ld r5, 0(r6)
j:              sd r3, 8(r7)
```

- **Register renaming** can be more easily done.
- Renaming can be done either statically by the compiler or dynamically by the hardware.

Data dependences and hazards

- A data/name dependence can potentially generate a data hazard (**RAW**, **WAW**, or **WAR**), but the actual hazard and the number of stalls to eliminate the hazards are a property of the pipeline.
  - **RAW** hazards correspond to true data dependences.
  - **WAW** hazards correspond to output dependences
  - **WAR** hazards correspond to antidependences.
- Dependences are a property of the program, while hazards are a property of the pipeline.
- Dependences
  - indicate the possibility of a hazard
  - determine the order in which results must be calculated
  - set an upper bound on parallelism

Control dependences

- A control dependence determines the ordering of instructions

```
if (p1) {
    S1;
 }
if (p2) {
    S2;
}
```

- Two constraints imposed by control dependences:
  - An instruction that is control dependent cannot be moved before the branch
  - An instruction that is not control dependent cannot be moved after the branch

Control dependences

- Control dependence is preserved by two properties:
  - Instructions execution in program order to ensure that an instruction that occurs before a branch is executed before the branch.
  - Detection of control hazards to ensure that an instruction (that is control dependent on a branch) is not executed until the branch direction is known.
- Although preserving control dependence is a simple way to preserve program order, **control dependence is not the critical property** that must be preserved.
- The two properties critical to program correctness are the exception behavior and data flow

Control dependences

- Preserving the exception behavior means that any change in ordering of instruction execution must not cause any new exceptions in the program

```
        DADDU R2,R3, R4
        BEQZ R2, L1
        LW R1,0(R2)
L1:
```

LW could generate an exception if R2=0 and LW is executed before the end of BEQZ

Control dependences

- Data flow is preserved by maintenance of data dependences and control dependences

```
        DADDU  R1, R2, R3
        BEQZ R4, L
        DSUBU R1,R5, R6
L:      …..
        OR R7, R1, R8
```

OR is executed after instruction L

OR uses the correct value of R1, i.e. when DADDU is execute before BEQZ and DSUBU is execute after BEQZ

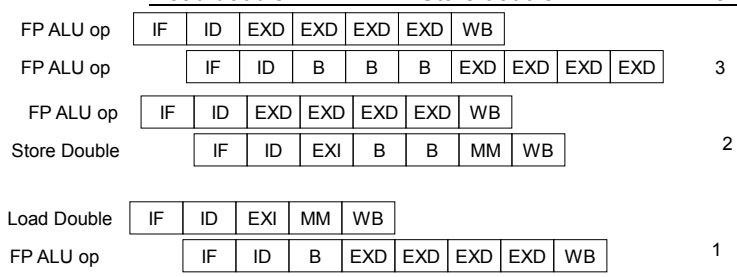
Instruction Level Parallelism

- **ILP = Exploit potential overlap of execution among unrelated instructions**
- Overlapping possible whenever:
  - No Structural Hazards
  - No RAW, WAR of WAW Hazards
  - No Control Hazards

Compiler Techniques for Exposing ILP

- To avoid a pipeline stall, a dependent instruction must be separated from the source instruction by a distance in clock cycles equal to the pipeline latency of that source instruction

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
Load double	Store double	0



Basic Pipeline Scheduling

```
double x[1000];
...
for (i=1; i<=1000; i++)
    x[i] = x[i] + s;
```

```
; r1 = address of the last element of x
; f2 = s
Loop:
  ld    f0,0(r1) ; f0=array element
  addd  f4,f0,f2 ; add scalar in f2
  sd    0(r1),f4 ; store result
  subi  r1,r1,8  ; descr. Pointer
  bnez  r1, Loop ; branch r1!=0
```

```
Loop:                          Clock cycles issued
  ld    f0,0(r1)               1
  stall                        2
  addd  f4,f0,f2               3
  stall                        4
  stall                        5
  sd    0(r1),f4               6
  subi  r1,r1,8                7
  stall                        8
  bnez  r1,Loop                9
  stall                        10
```

10 clock cycles per iteration

Basic Pipeline Scheduling

```
; r1 = address of the last element of x
; f2 = s
Loop:
  ld    f0,0(r1) ; f0=array element
  addd  f4,f0,f2 ; add scalar in f2
  sd    0(r1),f4 ; store result
  subi  r1,r1,8  ; descr. Pointer
  bnez  r1, Loop ; branch r1!=0
```

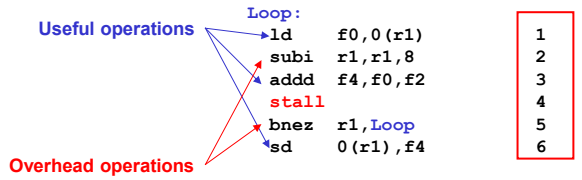
```
Loop:                          Clock cycles issued
  ld    f0,0(r1)               1
  stall                        2
  addd  f4,f0,f2               3
  stall                        4
  stall                        5
  sd    0(r1),f4               6
  subi  r1,r1,8                7
  stall                        8
  bnez  r1,Loop                9
  stall                        10
```

```
Loop:
  ld    f0,0(r1)
  subi  r1,r1,8
  addd  f4,f0,f2
  bnez  r1, Loop
  sd    8(r1),f4
```

```
Loop:                          Clock cycles issued
  ld    f0,0(r1)               1
  subi  r1,r1,8                2
  addd  f4,f0,f2               3
  stall                        4
  bnez  r1,Loop                5
  sd    0(r1),f4               6
```

6 clock cycles per iteration

Loop Unrolling



- We complete one loop iteration in 6 clock cycles, but the actual work of operating on the array element takes just 3 (the load, add, and store) of those 6 clock!
  - ➔ The remaining 3 clock cycles consist of loop overhead
- Since this loop will be executed many times, we can amortize the overhead over several iterations
  - ➔ **Loop Unrolling**

Loop Unrolling

- *Loop Unroll* so that there are 4 copies of the loop body (let's assume the number of loop iterations is a multiple of 4)

```
Loop:
  ld    f0,0(r1)
  addd  f4,f0,f2
  sd    0(r1),f4
  ld    f6,-8(r1)
  addd  f8,f6,f2
  sd    -8(r1),f8
  ld    f10,-16(r1)
  addd  f12,f10,f2
  sd    -16(r1),f12
  ld    f14,-24(r1)
  addd  f16,f14,f2
  sd    -24(r1),f16
  subi  r1,r1,32
  bnez  r1, Loop
```

1 stall
2 stalls
1 stall
2 stalls
1 stall
2 stalls
1 stall
2 stalls
1 stall
1 stall

14 stalls + 14 instructions = **28 clock cycles per iteration**

- Although this unrolled version is currently slower than the scheduled version of the original loop (7 cycles/element vs. 6 cycles/element), this will change when we schedule the unrolled loop
- Loop unrolled is normally done early in the compilation process

Loop Unrolling

- Unrolled loop after it has been scheduled on DLX

```
Loop:
  ld    f0,0(r1)
  ld    f6,-8(r1)
  ld    f10,-16(r1)
  ld    f14,-24(r1)
  addd  f4,f0,f2
  addd  f8,f6,f2
  addd  f12,f10,f2
  addd  f16,f14,f2
  sd    0(r1),f4
  sd    -8(r1),f8
  subi  r1,r1,32
  sd    16(r1),f12
  bnez  r1, Loop
  sd    8(r1),f16
```

0 stalls ➔ 14 clock cycles per iteration (3.5 cycles/ element)

	Clock cycles per iteration	Clock cycles per element
Original	10	10,0
Basic pipeline scheduling	6	6,0
Loop Unrolling	28	7,0
Loop Unrolling scheduled	14	3,5

Multiple-issue pipeline

- To reach higher performance (for a given technology) – more parallelism must be extracted from the program. In other words…**multiple-issue**
- Dependences must be detected and solved, and instructions must be *re-ordered* (***scheduled***) so as to achieve highest parallelism of instruction execution compatible with available resources.

Multiple-issue pipeline

- In a multiple-issue pipelined machine, the ideal CPI would be $CPI_{ideal} < 1$
- If we consider for example **2-issue** processor, *best case:* max throughput would be to complete 2 Instructions Per Clock: $IPC_{ideal} = 2$; $CPI_{ideal} = 0.5$

Getting CPI < 1

Issuing Multiple Instructions/Cycle

- 2-issue MIPS: 2 instructions, 1 FP & 1 anything else
  - Fetch 64-bits/clock cycle; Int on left, FP on right
  - Can only issue 2nd instruction if 1st instruction issues
  - More ports for FP registers to do FP load & FP op in a pair

Type	*Pipe Stages*						
Int. instruction	IF	ID	EX	MEM	WB		
FP instruction	IF	ID	EX	MEM	WB		
Int. instruction		IF	ID	EX	MEM	WB	
FP instruction		IF	ID	EX	MEM	WB	
Int. instruction			IF	ID	EX	MEM	WB
FP instruction			IF	ID	EX	MEM	WB

- 1 cycle load delay expands to 3 instructions in SS
  - ➔ Instruction in right half can't use it, nor instructions in next slot

Review: Unrolled Loop that Minimizes Stalls for Scalar

```
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     0(R1),F4
10      SD     -8(R1),F8
11      SD     -16(R1),F12
12      SUBI   R1,R1,#32
13      BNEZ   R1,LOOP
14      SD     8(R1),F16     ; 8-32 = -24
```

LD to ADDD: 1 Cycle
ADDD to SD: 2 Cycles

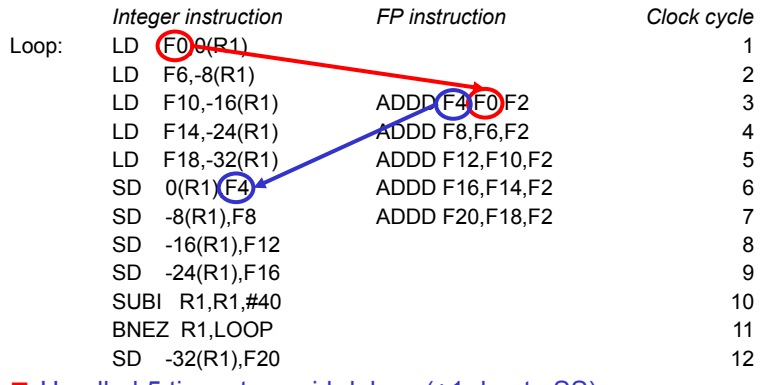
14 clock cycles per iteration
3.5 clock cycles per element

Unrolled Loop with 4 copies in 2–issue MIPS

```
LD F0,0(R1)          IF ID EXI MM  WB
NOP                  IF ID EX1 EX2 EX3 EX4 WB
LD F6,-8(R1)            IF ID EXI MM  WB
NOP                     IF ID  EX1 EX2 EX3 EX4 WB
LD F10,-16(R1)             IF  ID  EXI MM  WB
ADDD F4,F0,F2              IF  ID  EX1 EX2 EX3 EX4 WB
LD F14,-24(R1)                IF  ID EXI  MM  WB
ADDD F8,F6,F2                 IF  ID EX1 EX2 EX3 EX4 WB
SD 0(R1),F4                      IF  BB  ID EXI  MM  WB
ADDD F12,F10,F2                  IF  ID EX1 EX2 EX3 EX4 WB
```

Loop unroll with 4 copies requires 1 clock cycle stall

Loop Unrolling in 2-issue MIPS



	Integer instruction	*FP instruction*	*Clock cycle*
Loop:	LD F0,0(R1)		1
	LD F6,-8(R1)		2
	LD F10,-16(R1)	ADDD F4,F0,F2	3
	LD F14,-24(R1)	ADDD F8,F6,F2	4
	LD F18,-32(R1)	ADDD F12,F10,F2	5
	SD 0(R1),F4	ADDD F16,F14,F2	6
	SD -8(R1),F8	ADDD F20,F18,F2	7
	SD -16(R1),F12		8
	SD -24(R1),F16		9
	SUBI R1,R1,#40		10
	BNEZ R1,LOOP		11
	SD -32(R1),F20		12

- Unrolled 5 times to avoid delays (+1 due to SS)
- 12 clock cycles per iteration
- 2.4 clock cycles per element (1.5X)

Instruction Level Parallelism

- Two strategies to support ILP:
  - **Dynamic Scheduling:** Depend on the hardware to locate parallelism
  - **Static Scheduling:** Rely on software for identifying potential parallelism

Dynamic Scheduling

- The hardware reorder the instruction execution to reduce pipeline stalls while maintaining data flow and exception behavior.
- Main advantages (PROs):
  - It enables handling some cases where dependences are unknown at compile time
  - It simplifies the compiler complexity
  - It allows compiled code to run efficiently on a different pipeline (code portability).
- Those advantages are gained at a cost of (CONs):
  - a significant increase in hardware complexity,
  - power consumption.

Static Scheduling

- Compilers can use sophisticated algorithms for code scheduling to exploit **ILP (Instruction Level Parallelism).**
  - However the size of a **basic block** is usually quite **small** and the amount of parallelism available within a basic block is quite **small.**
  - Example: For typical MIPS programs the average branch frequency is between 15% and 25% => from 4 to 7 instructions execute between a pair of branches.

Static Scheduling

- Data dependence can further limit the amount of ILP we can exploit within a basic block to much less than the average basic block size.
- To obtain substantial performance enhancements, we must exploit ILP across multiple basic blocks.
- Static detection and resolution of dependences (**static scheduling**): accomplished by the **compiler** dependences are avoided by code reordering. Output of the compiler: reordered into dependency-free code.
- Typical example: **VLIW (Very Long Instruction Word)** processors expect dependency-free code generated by the compiler

Dynamic Scheduling

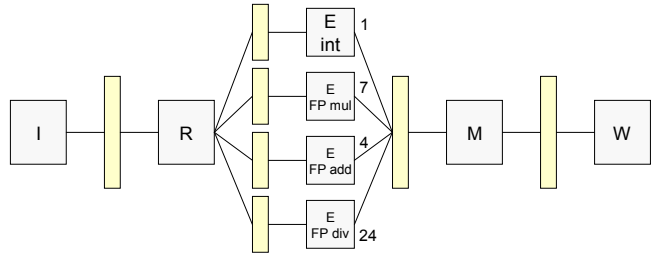
- Simple pipeline: hazards due to data dependences that cannot be hidden by forwarding *stall* the pipeline – no new instructions are fetched nor issued.
- ***Dynamic scheduling***: Hardware reorder instructions execution so as to reduce stalls, maintaining data flow and exception behaviour.
- ***Typical Example:*** *Superscalar Processor*

Dynamic Scheduling

- Basically: Instructions are *fetched* and *issued in program order* **(in-order-issue)**
- Execution begins ***as soon as operands are available*** – possibly, ***out of order execution*** – note: ***possible even with pipelined scalar architectures*** .
  - Out-of order execution introduces possibility of WAR, WAW data hazards.
- Out-of order execution implies ***out of order completion*** *unless there is a re-order buffer to get in-order completion*

Dynamic Scheduling: The Idea

- Major limitation: In-order instruction issue
  - ➔ If an instruction is stalled in the pipeline, no later instructions can proceed
  - ➔ If there are multiple functional units, these units could lie idle




```
divd f0,f2,f4
addd f10,f0,f8
subd f12,f8,f14
```

- The `subd` cannot execute because the dependence of `addd` on `divd` causes the pipeline stall
- `subd` is not data dependent on anything in the pipeline
- This is a performance limitation that can be eliminated by not requiring instructions to execute in order

Dynamic Scheduling: The Idea

- Dynamic instruction scheduling attemps to exploit ILP by allowing instructions to execute as early as possible when
  - ➔ There is an available functional unit and no pending destination
    - ✓ To avoid structural hazards and WAW hazards
  - ➔ Source operands are available
    - ✓ To avoid RAW hazards
  - ➔ To write results when destination registers are no longer needed
    - ✓ To avoid WAR hazards

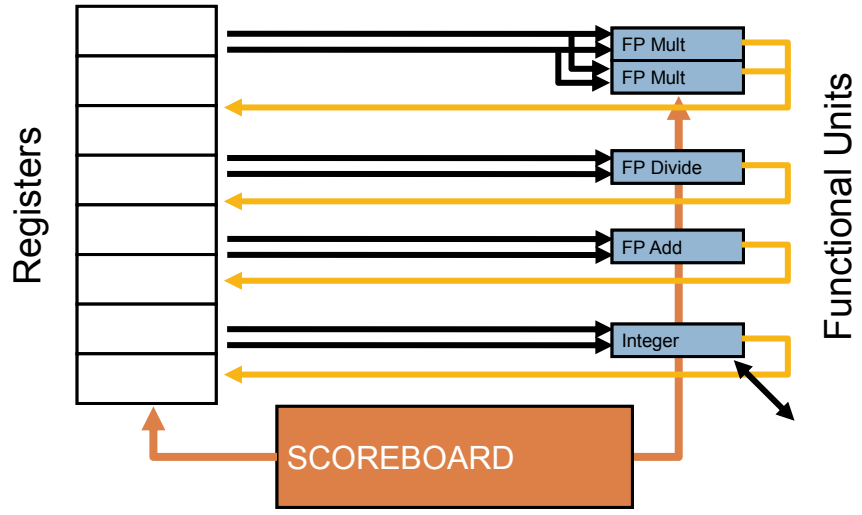
```
divd   f0,f2,f4          divd   f0,f2,f4
addd   f10,f0,f8         addd   f10,f0,f8
subd   f8,f8,f14         subd   f10,f8,f14
```

WAR hazard

WAW hazard

Scoreboard Dynamic Scheduling Algorithm

Scoreboard Architecture (CDC 6600)



Dynamic Scheduling with a Scoreboard

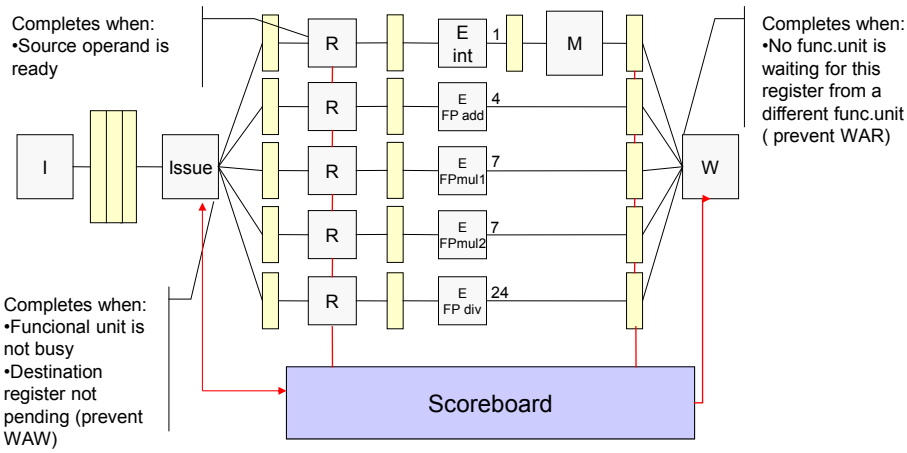
- Out-of-order execution divides ID stage
  1. **Issue**—decode instructions, check for structural hazards
  2. **Read operands**—wait until no data hazards, then read operands
- Scoreboards date to CDC 6600 in 1963
- Instructions execute whenever not dependent on previous instructions and no hazards
- CDC 6600: In order issue, out-of-order execution, out-of-order commit (or completion)
  - ➔ No forwarding!

Scoreboard basic scheme

- We distinguish when an instruction begins execution and it completes execution: between the two times, the instruction is ***in execution.***
- We assume the pipeline allows multiple instructions in execution at the same time that requires multiple functional units, pipelined functional units or both.
- **CDC 6600: In order issue, out of order execution, out of order completion (commit)**

Dynamic Scheduling with a Scoreboard

- The idea of a scoreboard is to keep track of the status of instructions, functional units and registers



Scoreboard basic scheme

- Scoreboard replaces ID, EX, WB stages with 4 stages
- ID stage split in two parts:
  - **Issue** (decode and check structural hazard)
  - **Read Operands** (wait until no data hazards)
- Scoreboard allows instructions **without dependencies** to execute
- **In-order *issue* BUT out-of-order *read-operands* *out-of-order execution and completion***
- All instructions pass through the issue stage in-order, but they can be stalled or bypass each other in the read operand stage and thus enter execution out-of-order and with different latencies, which implies out-of-order completion

Scoreboard Implications

- Out-of-order completion ➔ WAR and WAW hazards can occur
- Solutions for WAR:

```
divd    f0,f2,f4
addd    f10,f0,f8
subd    f8,f8,f14
```

  - ➔ Stall writeback until registers have been read
  - ➔ Read registers only during Read Operands stage
- Solution for WAW:

```
divd    f0,f2,f4
addd    f10,f0,f8
subd    f10,f8,f14
```

  - ➔ Detect hazard and stall issue of new instruction until other instruction completes
- Scoreboard keeps track of dependencies between instructions that have already issued

Scoreboard scheme

- **Hazard detection and resolution is centralized in the scoreboard:**
  - Every instruction goes through the Scoreboard, where a record of data dependences is constructed
  - The Scoreboard then determines when the instruction can read its operand and begin execution
  - If the scoreboard decides the instruction cannot execute immediately, it monitors every change and decides when the instruction can execute.
  - The scoreboard controls when the instruction can write its result into destination register

Four Stages of Scoreboard Control

- **Issue**—decode instructions & check for structural hazards (ID1)
  - ➔ Instructions issued in program order (for hazard checking)
    - ➔ If a functional unit for the instruction is free and no other active instruction has the same destination register (no WAW), the scoreboard issues the instruction to the functional unit and updates its internal data structure.
  - ➔ Don't issue if structural hazard or WAW hazard
  - ➔ Don't issue if instruction is output dependent on any previously issued but uncompleted instruction (no WAW hazards)

Four Stages of Scoreboard Control

- **Read operands**—wait until no data hazards, then read operands (ID2)
  - A source operand is available if:
    - No earlier issued active instruction will write it or
    - A functional unit is writing its value in a register
  - When the source operands are available, the scoreboard tells the functional unit to proceed to read the operands from the registers and begin execution.
  - RAW hazards are resolved dynamically in this step, and instructions may be sent into execution out of order.
  - No forwarding of data in this model!

Four Stages of Scoreboard Control

- **Execution**—operate on operands (EX)
  - ➔ The functional unit begins execution upon receiving operands
  - ➔ When the result is ready, it notifies the scoreboard that it has completed execution

- FUs are characterized by:
  - **Variable latency** (the effective time used to complete one operation).
  - Initiation interval (the number of cycles that must elapse between issuing two operations to the same functional unit).
  - Load/Store latency depends on data cache HIT/MISS

Four Stages of Scoreboard Control

- **Write result**—finish execution (WB)
  - Stall until no WAR hazards with previous instructions:

Example:

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

CDC 6600 scoreboard would stall SUBD until ADDD reads operands

CDC 6600 Scoreboard

- Limitations of 6600 scoreboard:
  - No forwarding hardware
  - Limited to instructions in basic block (small *window*)
  - Small number of functional units (structural hazards), especially integer/load store units
  - Do not issue on structural hazards
  - Wait for WAR hazards
  - Prevent WAW hazards

Three Parts of the Scoreboard

- **Instruction status**
  Which of 4 steps the instruction is in

- **Functional unit status**
  - ➔ Indicates the state of the functional unit (FU). 9 fields for each functional unit
    - ✓ Busy: Indicates whether the unit is busy or not
    - ✓ Op: Operation to perform in the unit (e.g., + or –)
    - ✓ Fi: Destination register
    - ✓ Fj,Fk: Source-register numbers
    - ✓ Qj,Qk: Functional units producing source registers Fj, Fk
    - ✓ Rj,Rk: Flags indicating when Fj, Fk are ready

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

Scoreboard Example

Instruction status:

Instruction		j	k	Issue	Read Oper	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				

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status:

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

Scoreboard Example: Cycle 1

Instruction status:

Instruction		*j*	*k*	*Issue*	*Read Oper*	*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				

Functional unit status:

Time	*Name*	*Busy*	*Op*	*dest Fi*	*S1 Fj*	*S2 Fk*	*FU Qj*	*FU Qk*	*Fj? Rj*	*Fk? Rk*
	Integer	Yes	Load	F6		R2				Yes
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status:

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

Scoreboard Example: Cycle 2

Instruction status:

Instruction		*j*	*k*	*Issue*	*Read Oper*	*Exec Comp*	*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				

- **Issue 2nd LD?**

Functional unit status:

Time	*Name*	*Busy*	*Op*	*dest Fi*	*S1 Fj*	*S2 Fk*	*FU Qj*	*FU Qk*	*Fj? Rj*	*Fk? Rk*
	Integer	Yes	Load	F6		R2				Yes
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status:

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

Scoreboard Example: Cycle 3

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	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				

- **Issue MULT?**

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	Yes	Load	F6		R2				No
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
3	FU				Integer					

Scoreboard Example: Cycle 4

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	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	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
4	FU				Integer					

Scoreboard Example: Cycle 5

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	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	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	Yes	Load	F2		R3				Yes
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
5	FU		Integer							

Scoreboard Example: Cycle 6

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	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	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	Yes	Load	F2		R3				Yes
	Mult1	Yes	Mult	F0	F2	F4	Integer		No	Yes
	Mult2	No								
	Add	No								
	Divide	No								

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
6	FU	Mult1	Integer							

Scoreboard Example: Cycle 7

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	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	S1 Fj	S2 Fk	FU Qj	FU 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:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
7	FU	Mult1	Integer			Add				

Scoreboard Example: Cycle 8a

(First half of clock cycle)

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	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	8			
ADDD	F6	F8	F2				

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU 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	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
8	FU	Mult1	Integer			Add	Divide			

Scoreboard Example: Cycle 8b

(First half of clock cycle)

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	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	S1 Fj	S2 Fk	FU Qj	FU 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:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
8	FU	Mult1				Add	Divide			

Scoreboard Example: Cycle 9

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	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	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
10	Mult1	Yes	Mult	F0	F2	F4			Yes	Yes
	Mult2	No								
2	Add	Yes	Sub	F8	F6	F2			Yes	Yes
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Note: Remaining

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
9	FU	Mult1				Add	Divide			

Scoreboard Example: Cycle 10

Instruction status:

Instruction		*j*	*k*	*Issue*	*Read Oper*	*Exec Comp*	*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				

Functional unit status:

Time	*Name*	*Busy*	*Op*	*dest Fi*	*S1 Fj*	*S2 Fk*	*FU Qj*	*FU Qk*	*Fj? Rj*	*Fk? Rk*
	Integer	No								
9	Mult1	Yes	Mult	F0	F2	F4			No	No
	Mult2	No								
1	Add	Yes	Sub	F8	F6	F2			No	No
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
10	*FU*	Mult1				Add	Divide			

Scoreboard Example: Cycle 11

Instruction status:

Instruction		*j*	*k*	*Issue*	*Read Oper*	*Exec Comp*	*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				

Functional unit status:

Time	*Name*	*Busy*	*Op*	*dest Fi*	*S1 Fj*	*S2 Fk*	*FU Qj*	*FU 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:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
11	*FU*	Mult1				Add	Divide			

Scoreboard Example: Cycle 12

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	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	S1 Fj	S2 Fk	FU Qj	FU 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:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
12	*FU*	Mult1					Divide			

Scoreboard Example: Cycle 13

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	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			

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU 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:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
13	*FU*	Mult1			Add		Divide			

Scoreboard Example: Cycle 14

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	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		

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
5	Mult1	Yes	Mult	F0	F2	F4			No	No
	Mult2	No								
2	Add	Yes	Add	F6	F8	F2			Yes	Yes
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
14	FU	Mult1			Add		Divide			

Scoreboard Example: Cycle 15

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	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		

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
4	Mult1	Yes	Mult	F0	F2	F4			No	No
	Mult2	No								
1	Add	Yes	Add	F6	F8	F2			No	No
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
15	FU	Mult1			Add		Divide			

Scoreboard Example: Cycle 16

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	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	

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU 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:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
16	*FU*	Mult1			Add		Divide			

Scoreboard Example: Cycle 17

Instruction status:

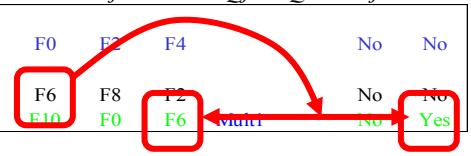
Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9		
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2	13	14	16	

- Why not write result of ADD???

WAR Hazard!

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU 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:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
17	*FU*	Mult1			Add		Divide			

Scoreboard Example: Cycle 18

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	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	

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
1	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:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
18	*FU*	Mult1			Add		Divide			

Scoreboard Example: Cycle 19

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	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	
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	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
0	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:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
19	*FU*	Mult1			Add		Divide			

Scoreboard Example: Cycle 20

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	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	S1 Fj	S2 Fk	FU Qj	FU 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:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
20	*FU*				Add		Divide			

Scoreboard Example: Cycle 21

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	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	

- **WAR Hazard is now gone...**

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU 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:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
21	*FU*				Add		Divide			

Scoreboard Example: Cycle 22

Instruction status:

Instruction		j	k	*Issue*	*Read Oper*	*Exec Comp*	*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

Functional unit status:

Time	*Name*	*Busy*	*Op*	*dest Fi*	*S1 Fj*	*S2 Fk*	*FU Qj*	*FU Qk*	*Fj? Rj*	*Fk? Rk*
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
39	Divide	Yes	Div	F10	F0	F6			No	No

Register result status:

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

Faster than light computation (skip a couple of cycles)

Scoreboard Example: Cycle 61

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	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	
ADDD	F6	F8	F2	13	14	16	22

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
0	Divide	Yes	Div	F10	F0	F6			No	No

Register result status:

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

Scoreboard Example: Cycle 62

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	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	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status:

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

Review: Scoreboard Example: Cycle 62

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	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

- **In-order issue; out-of-order execute & commit**

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status:

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