

Lecture 18

Instruction Level Parallelism

Today's lecture

- Pipelining and other forms of Instruction Level Parallelism (ILP)
- Building an efficient barrier

What makes a processor run faster?

- Registers and cache
- Vectorization (SSE)
- *Instruction level parallelism*

What is Instruction-level Parallelism?

- The potential to execute certain instructions in parallel, because they are independent

$w = u / z;$

$a = b + c;$

- Any technique for identifying and exploiting such opportunities
 - Static: can be implemented by a compiler
 - Dynamic: requires hardware support

How does ILP manifest itself?

- Basic blocks
 - Sequences of instruction that appear between branches
 - Usually no more than 5 or 6 instructions!
- Loops

```
for ( i=0; i < N; i++)  
    x[i] = x[i] + s;
```
- We can only realize ILP by finding sequences of independent instructions
- Dependent instructions must be separated by a sufficient amount of time, that is functional unit latency

Pipelining

- An implementation technique for overlapping instruction execution
- Like an automobile assembly line, we break the instruction's execution into steps
- We can then overlap the steps so long as there are no dependences (*pipeline hazards*) between the steps

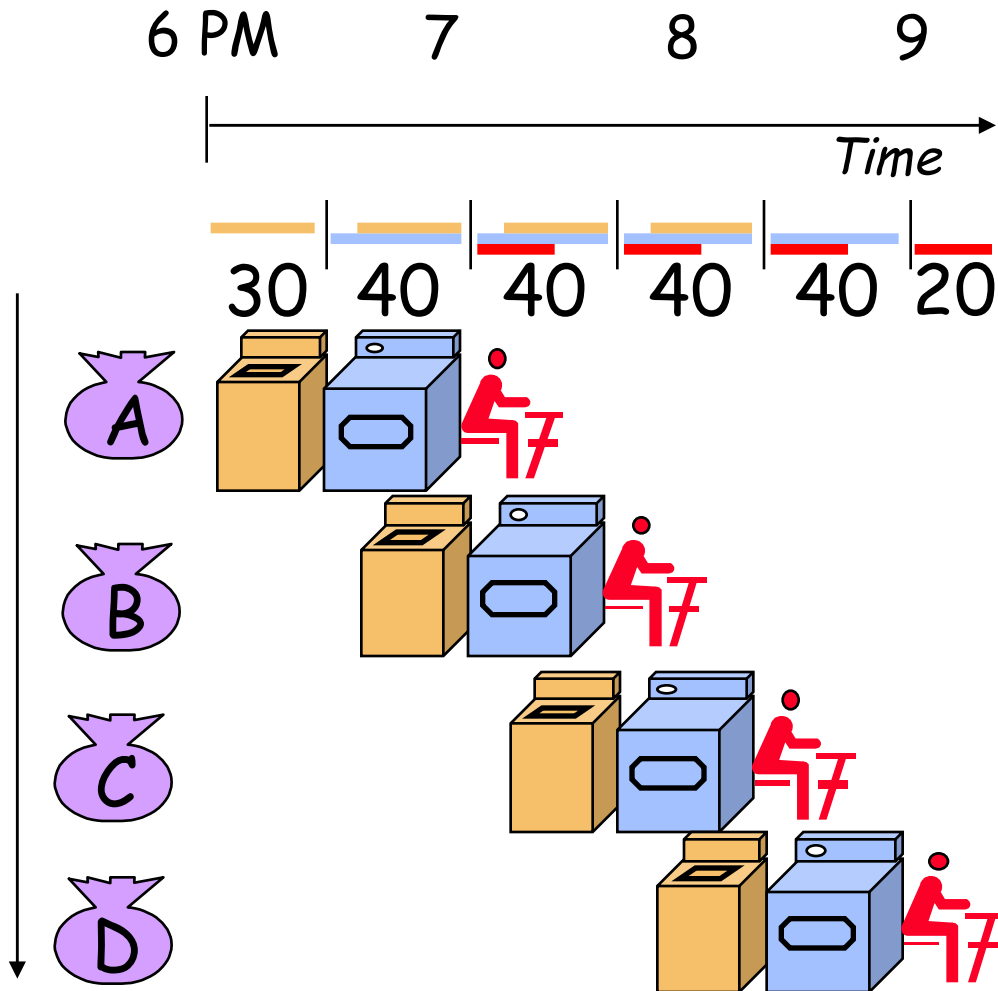


www.ford.ie

Pipelining

- Laundry analogy

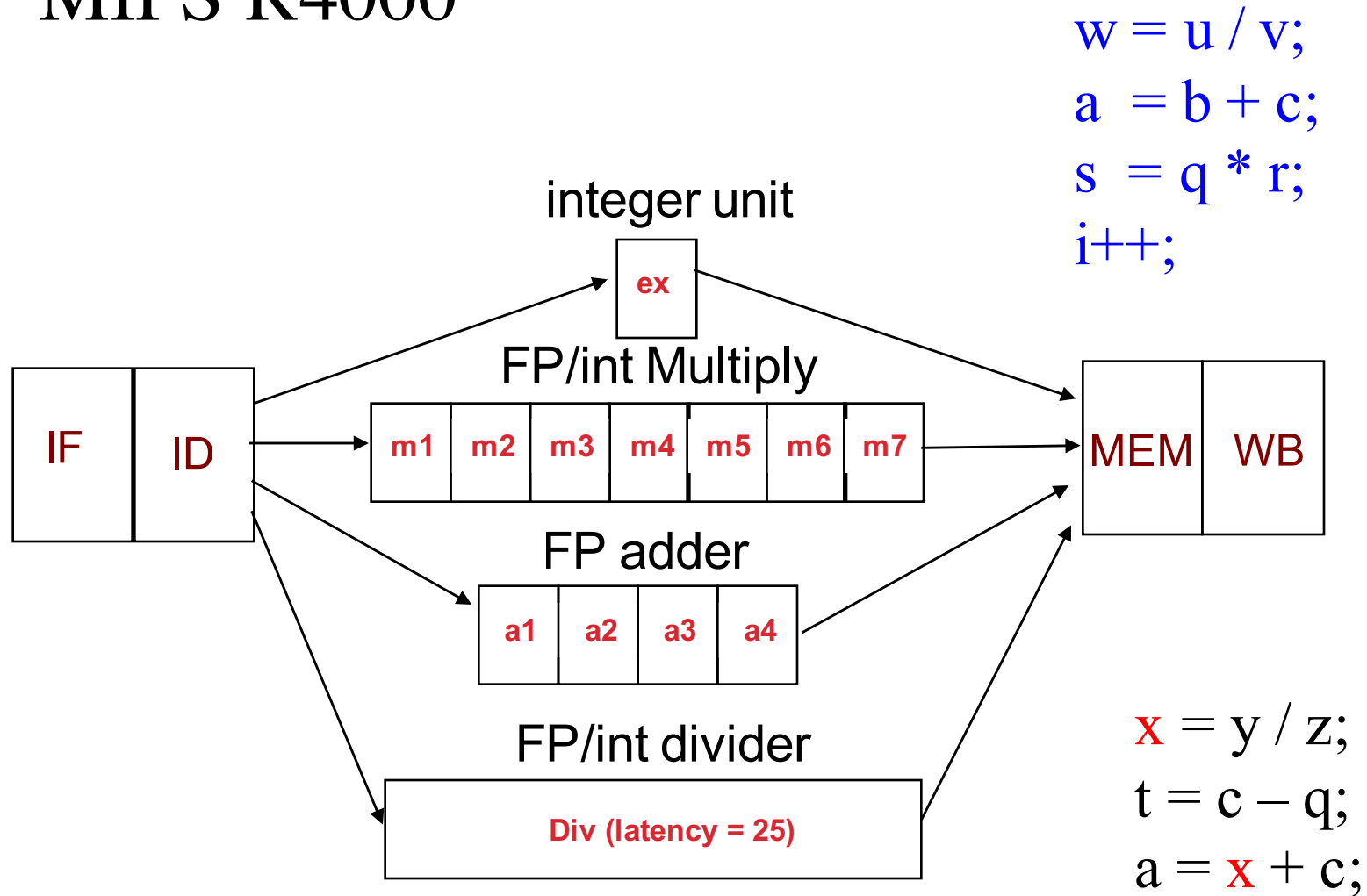
wash (30 min) + dry (40 min) + fold (20 min) = 90 min



- Sequential execution takes $4 * 90 \text{ min} = 6 \text{ hours}$
- Pipelined execution takes $30 + 4 * 40 + 20 = 3.5 \text{ hours}$
- Bandwidth = loads/hour
- Pipelining helps bandwidth but not latency (90 min)
- Bandwidth limited by slowest pipeline stage
- Potential speedup = Number pipe stages

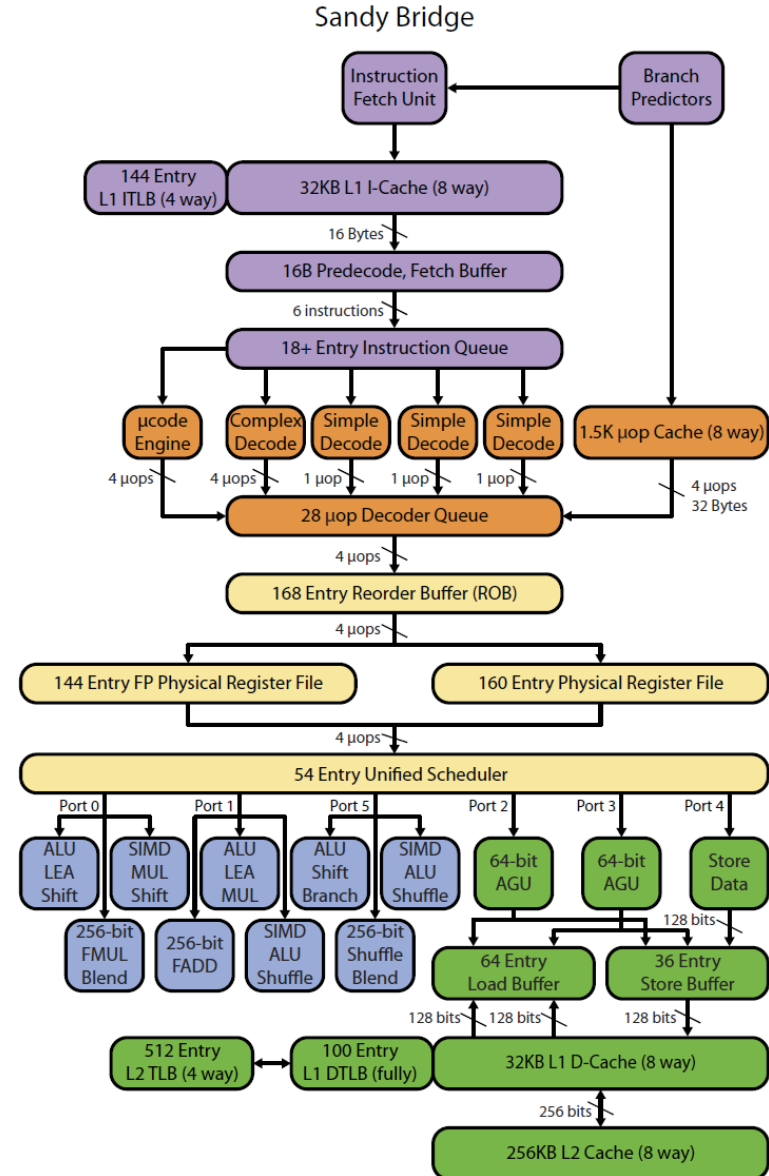
Multi-execution pipeline

- MIPS R4000



David Culler

Sandy Bridge



Pipeline Hazards

- What if we have dependencies?
- A data dependence is one kind of pipeline *hazard*, a *data hazard*, that prevents the next instruction from executing when expected
- Structural hazards: physical resources aren't available
- Control hazards: branches or other instructions that change the PC
- The simplest solution is to *stall* the pipeline, introducing *bubble*, but we can do better

$x = y / z;$
 $a = x + c;$
 $t = c - q;$

Limitations of ILP

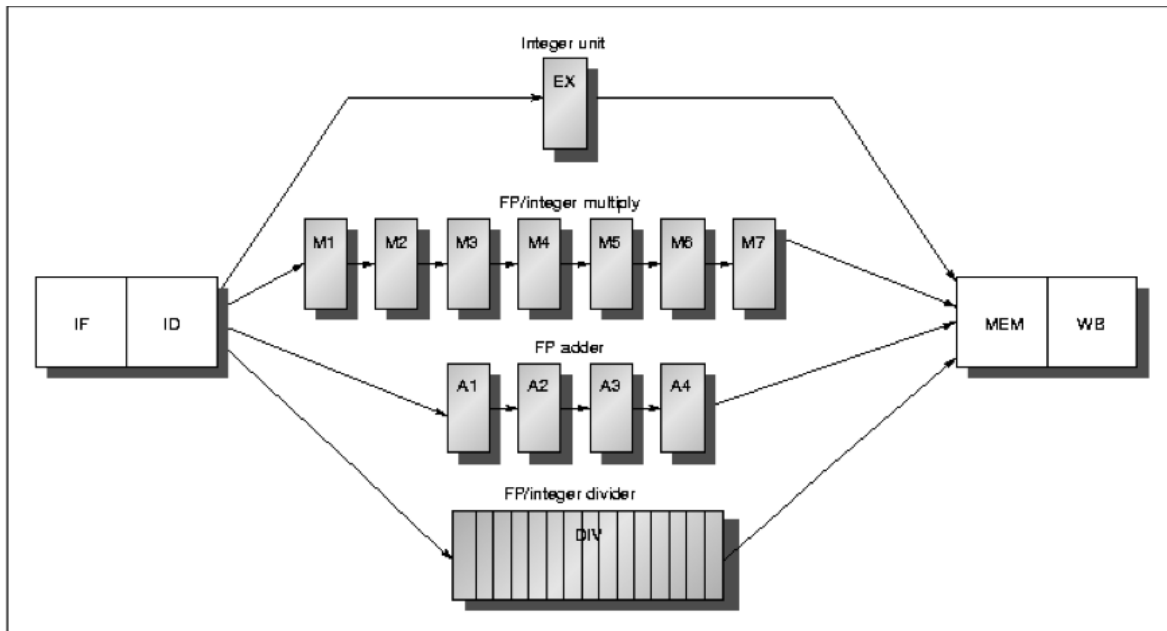
- Consider this loop
for (i=0; i<N; i++)
x[i] = x[i] + s;

L:LD	F0, 0(R1)	; F0 is the vector element
ADDD	F4, F0, F2	; add the scalar
SD	0(R1), F4	; store the result
SUBI	R1, R1, #8	; decrement by 8
		; to previous word
BNEZ	R1, L	; Branch if R1 $\neq$ 0
NOP		; Delayed branch slot

Delays

```
for ( i=0; i<N; i++)
    x[i] = x[i] + s;
```

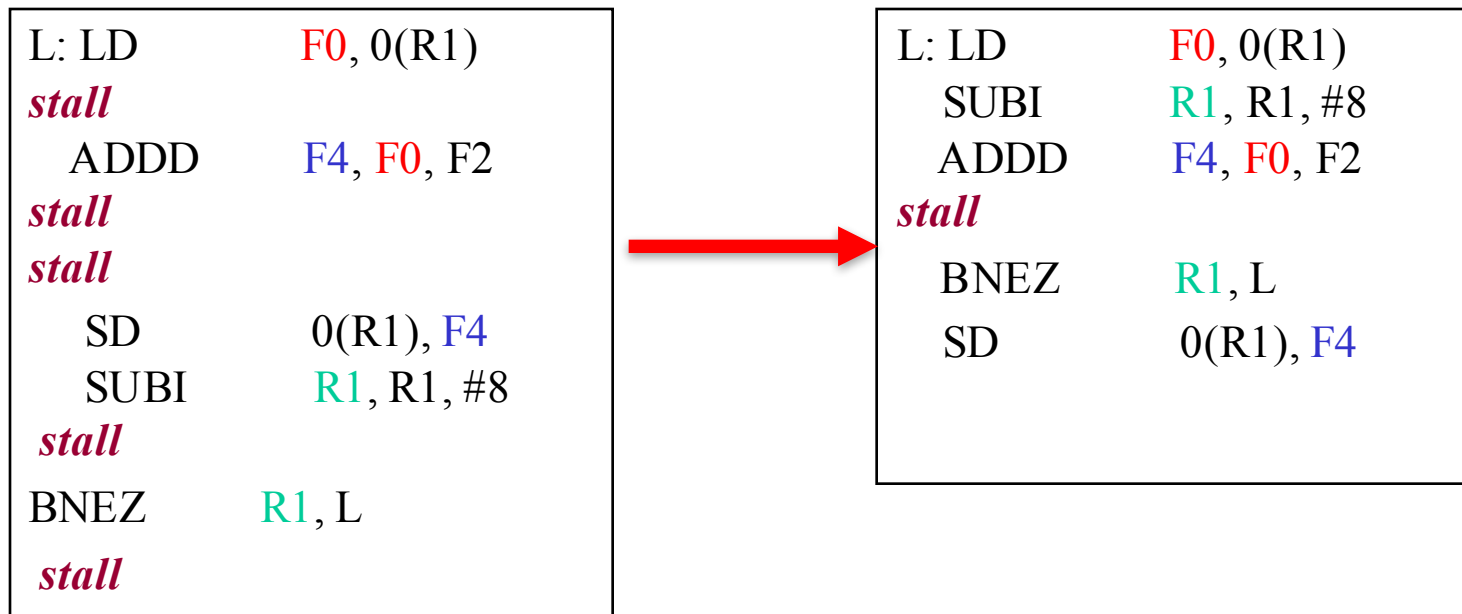
Unit	Latency	Init Interval
Integer	0	1
Memory	1	1
FP Add	3	1
FP Multiply	6	1
FP Div	24	24



L: LD F0, 0(R1)
stall
 ADDD F4, F0, F2
stall
stall
 SD 0(R1), F4
 SUBI R1, R1, #8
stall
 BNEZ R1, L
stall

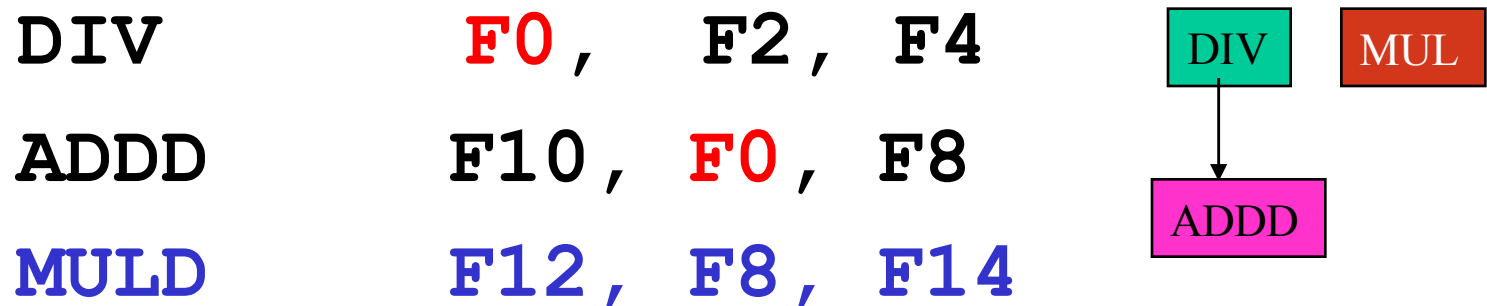
Instruction Reordering Reduces stalls

```
for ( i=0; i<N; i++)  
    x[i] = x[i] + s;
```



Static scheduling limits performance

- We can't always reorder, or even if it were possible there is a limit to how far the compiler can look
- The ADDD instruction is stalled on the DIVide ...
- ...stalling further instruction issue, e.g. the SUBD

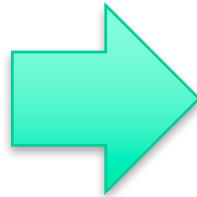


- But MULD doesn't depend on ADDD or DIV
- If we have an adder and a multiplier unit, one will sit idle uselessly until the DIV finishes

Dynamic scheduling

- Idea: Modify the pipeline to enable instructions to execute as soon as their operands become available
- This is known as *out-of-order execution*
- Stalled instructions (MUL) can now proceed normally

```
x = y / z;  
a = x + c;  
t = c * q;
```

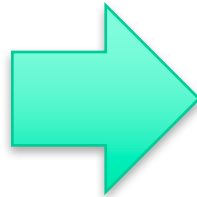


```
x = y / z;  
t = c * q;  
a = x + c;
```

Complications of dynamic scheduling

- Dynamically scheduled instructions also complete out of order
- Increased hardware complexity
 - Stations and execution units
 - Bookkeeping
 - Buffering
 - Slows down the processor

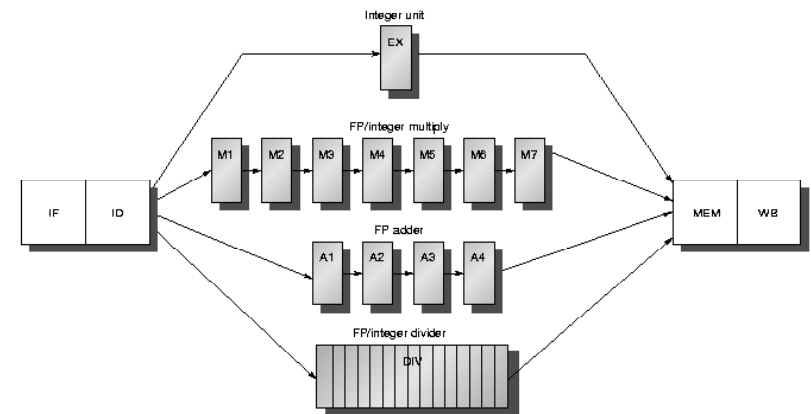
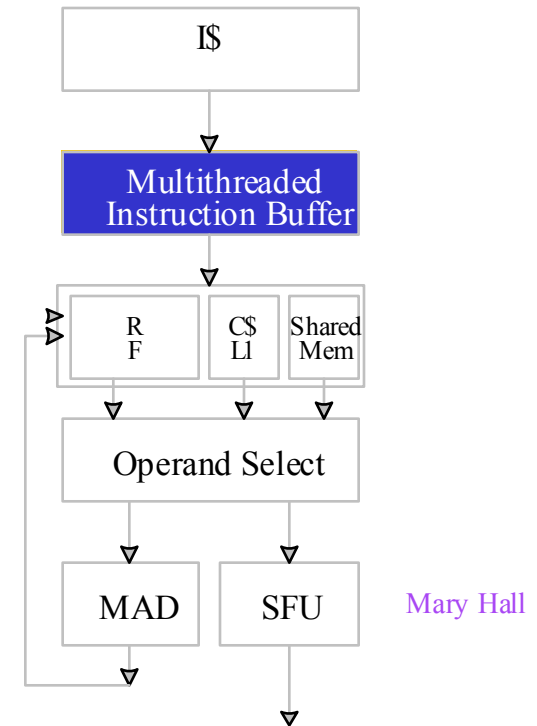
```
x = y / z;  
a = x + c;  
t = c * q;
```



```
x = y / z;  
t = c * q;  
a = x + c;
```

Dynamic scheduling splits the ID stage

- Issue sub-stage
 - Decode the instructions
 - Check for structural hazards
- Read operands sub-stage
 - Wait until there are no data hazards
 - Read operands



Consequences of a split ID stage

- We distinguish between the time when an instruction **begins** execution, and when it **completes**
- Previously, an instruction stalled in the ID stage, and this held up the entire pipeline
- Instructions can now be in a suspended state, neither stalling the pipeline, nor executing
- They are waiting on operands - need additional registers to store pending instructions that aren't ready to execute

The three steps of instruction execution

- Issue
 - ▶ Get instruction from fetch queue
 - ▶ If there are available resources buffer the register operands at station
 - ▶ If there is no such station, stall the instruction
- Execute
 - ▶ Monitor internal interconnect for any need operands
 - ▶ Intercept those operands and write to the station
 - ▶ When both operands are present, execute the operation
- Write result
 - ▶ Send to the result registers and any waiting units

Automatic, dynamic loop unrolling

```
L:  LD      F0, 0(R1)
    MULTD   F4, F0, F2
    SD      0(R1), F4
    SUBI    R1, R1, #8
    BNEZ    R1, L
```

Automatic, dynamic loop unrolling

Instruction		Iteration	Issue Exec
LD	F0, 0(R1)	1	X

Automatic, dynamic loop unrolling

Instruction		Iteration	Issue	Exec
LD	F0, 0(R1)	1	X	X
MULTD	F4, F0, F2	1	X	

Automatic, dynamic loop unrolling

Instruction		Iteration	Issue	Exec
LD	F0, 0(R1)	1	X	X
MULTD	F4, F0, F2	1	X	
SD	0(R1), F4	1	X	

Automatic, dynamic loop unrolling

Instruction		Iteration	Issue	Exec
LD	F0, 0(R1)	1	X	X
MULTD	F4, F0, F2	1	X	
SD	0(R1), F4	1	X	
LD	F0, 0(R1)	2	X	

Automatic, dynamic loop unrolling

Instruction		Iteration	Issue	Exec
LD	F0, 0(R1)	1	X	X
MULTD	F4, F0, F2	1	X	
SD	0(R1), F4	1	X	
LD	F0, 0(R1)	2	X	X

Automatic, dynamic loop unrolling

Instruction		Iteration	Issue	Exec
LD	F0, 0(R1)	1	X	X
MULTD	F4, F0, F2	1	X	
SD	0(R1), F4	1	X	
LD	F0, 0(R1)	2	X	X
MULTD	F4, F0, F2	2	X	
SD	0(R1), F4	2	X	

Two schemes for dynamic scheduling

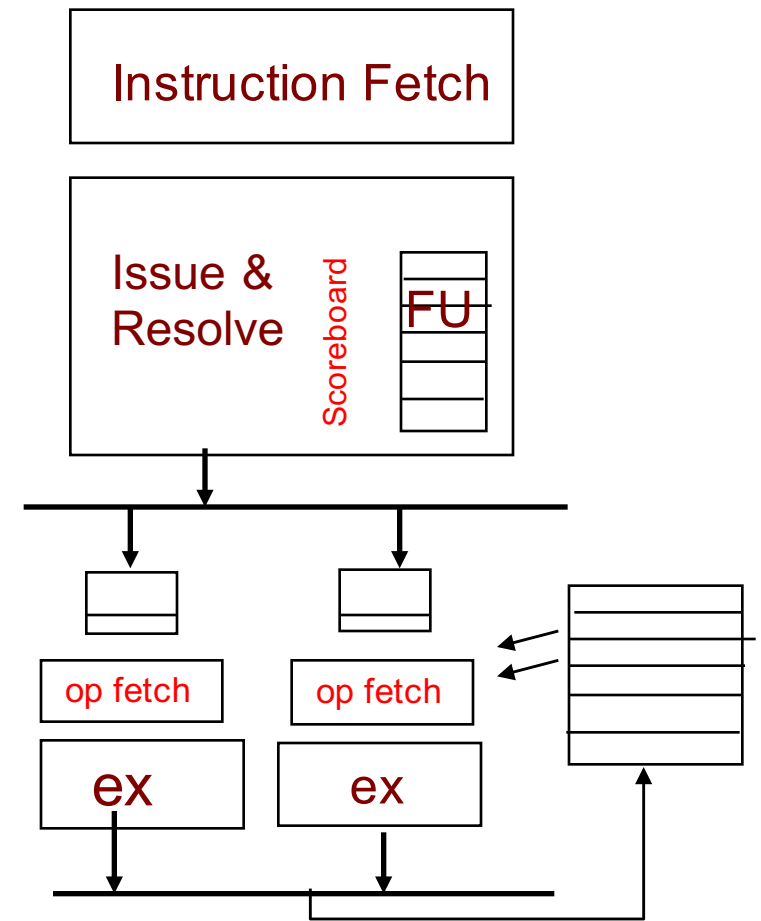
- Scoreboard
 - CDC 66000
- Tomasulo's algorithm
 - IBM 360/91
- We'll vary the number of functional units, their latency, and functional unit pipelining

What is a scoreboarding?

- A technique that allows instructions to execute out of order...
 - So long as there are sufficient resources and
 - No data dependencies
- The goal of scoreboarding
 - Maintain an execution rate of one instruction per clock cycle

Scoreboarding Strategy

- Hardware data structures keep track of
 - ▶ When instructions complete
 - ▶ Which instructions depend on the results
 - ▶ When it's safe to write a reg.
- Deals with data hazards
 - ▶ **WAR** (Write after read)
 - ▶ **RAW** (Read after write)
 - ▶ **WAW** (Write after write)



$a = b * c$
 $x = y - b$
 $q = a / y$
 $y = x + b$

Where do RAW hazards appear?

- A. (2)
- B. (3)
- C. (4)
- D. (2) and (3)
- E. (3) and (4)

```
(1) a = b * c
(2) x = y - b
(3) q = a / y
(4) y = x + b
```

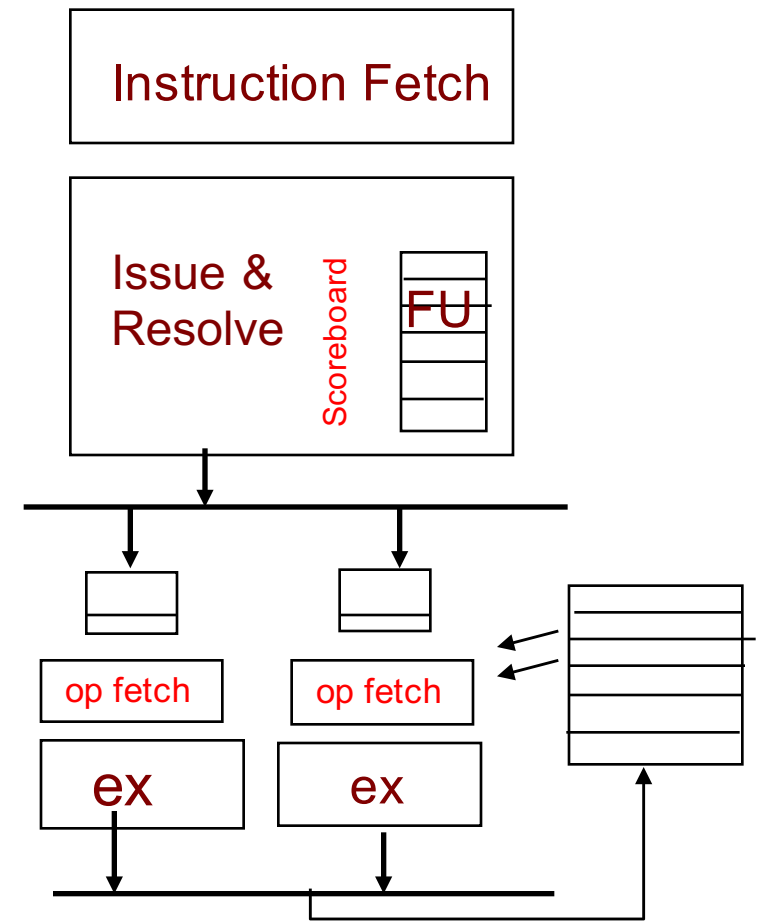
Where do WAR hazards appear?

- A. (2)
- B. (3)
- C. (4)
- D. (2) and (3)
- E. (2), (3) and (4)

```
(1) a = b * c
(2) x = y - b
(3) q = a / y
(4) y = x + b
```

Hazards

- ▶ **WAR** (Write after read)
- ▶ **RAW** (Read after write)
- ▶ **WAW** (Write after write)



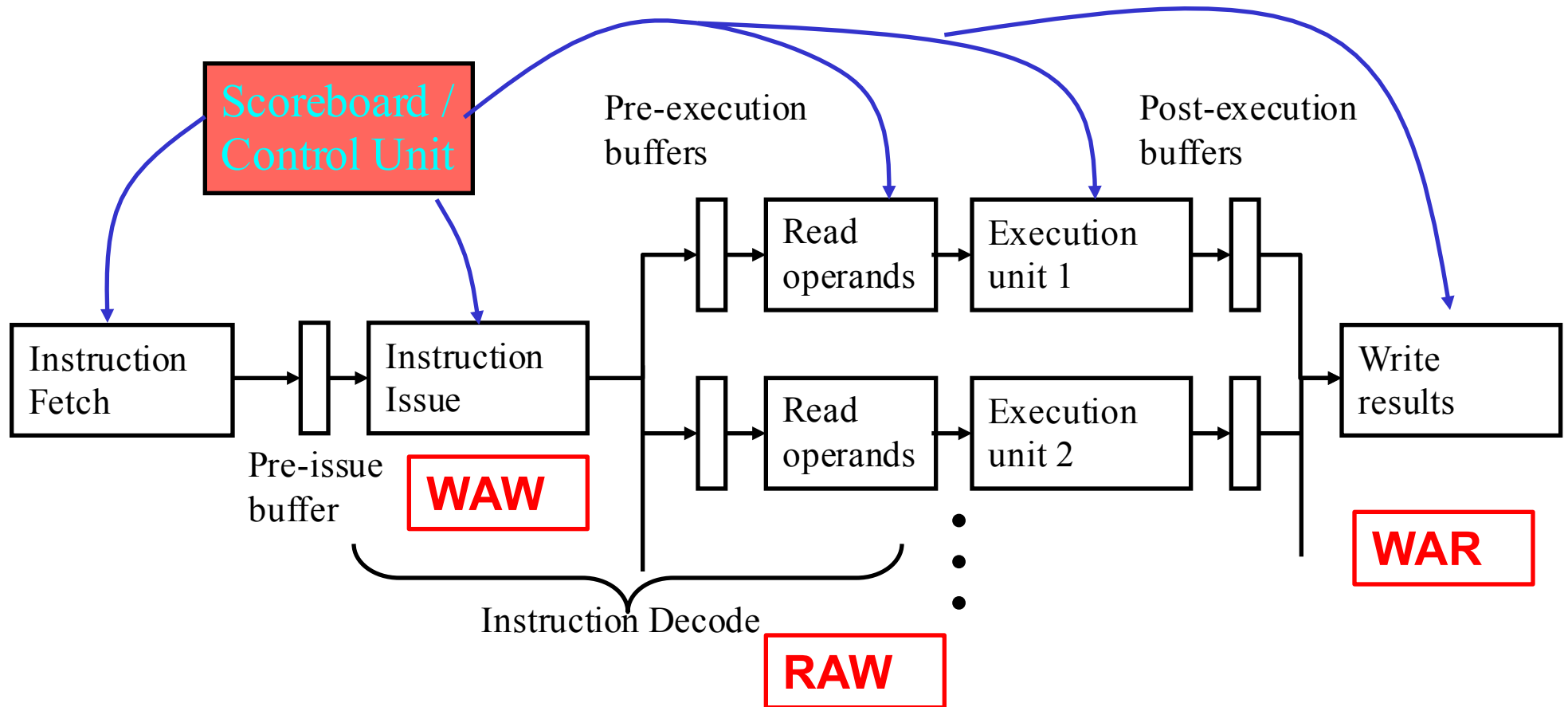
$a = b * c$

$x = y - b$

$q = a / y$

$y = x + b$

Scoreboard controls the pipeline



Mike Frank

What are the requirements?

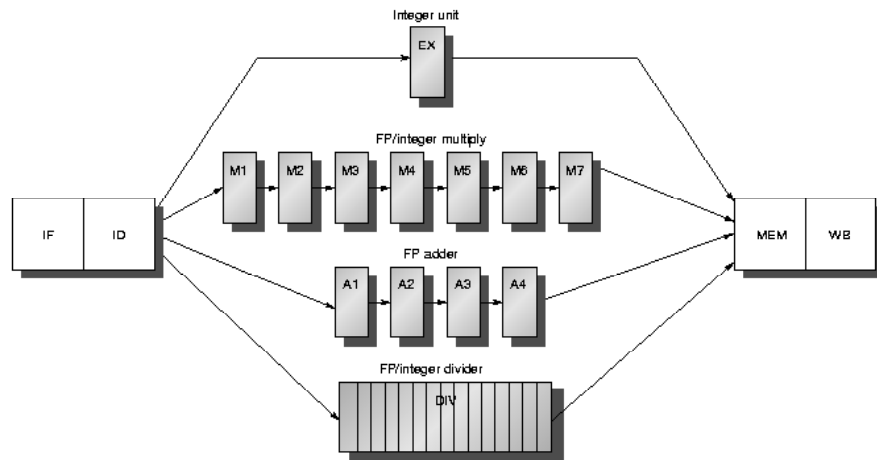
- Responsibility for instruction issue and execution, including hazard detection
- Multiple instructions must be in the EX stage simultaneously ...
- ... either through pipelining or multiple functional units
- Our processor (DLX) has:
2 multipliers, 1 divider, 1 integer unit
(memory, branch, integer arithmetic)

How does it work?

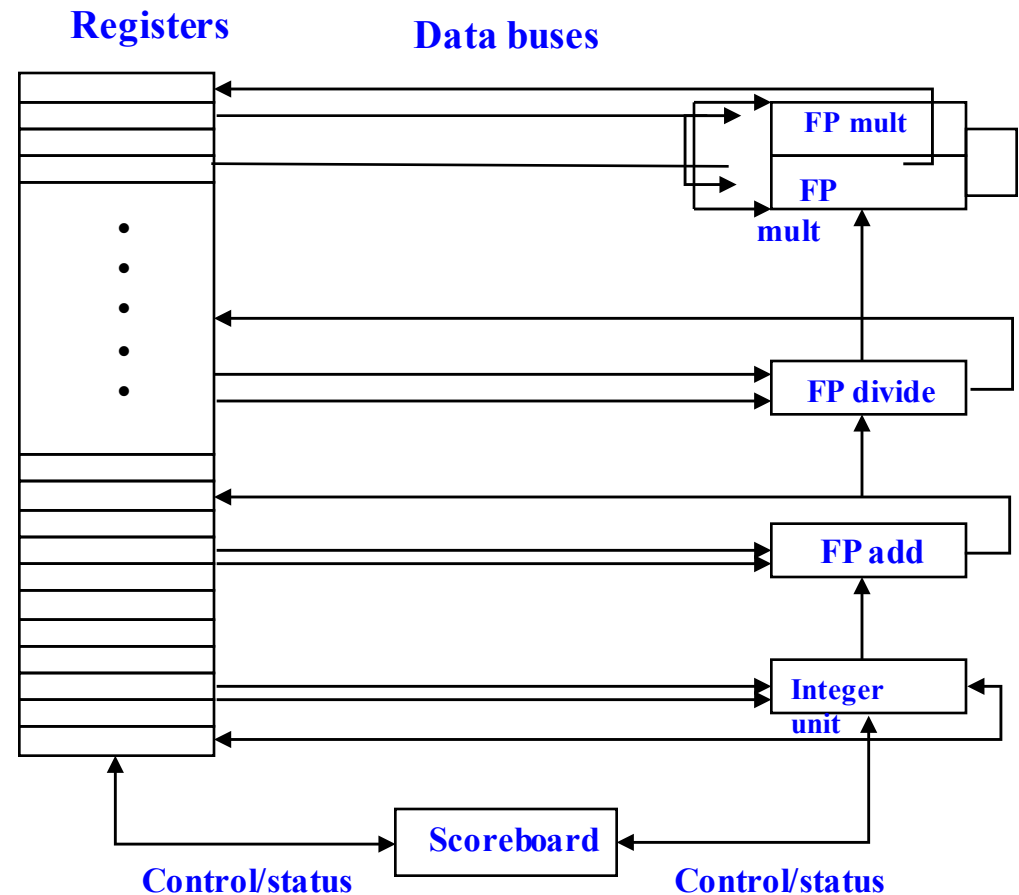
- As each instruction passes through the scoreboard, construct a description of the data dependencies (Issue)
- Scoreboard determines when the instruction can read operands and begin execution
- If the instruction can't begin execution, the scoreboard keeps a record, and it listens for one the instruction *can* execute
- Also controls when an instruction may write its result
- All hazard detection is centralized

Multiple execution pipelines in DLX with scoreboarding

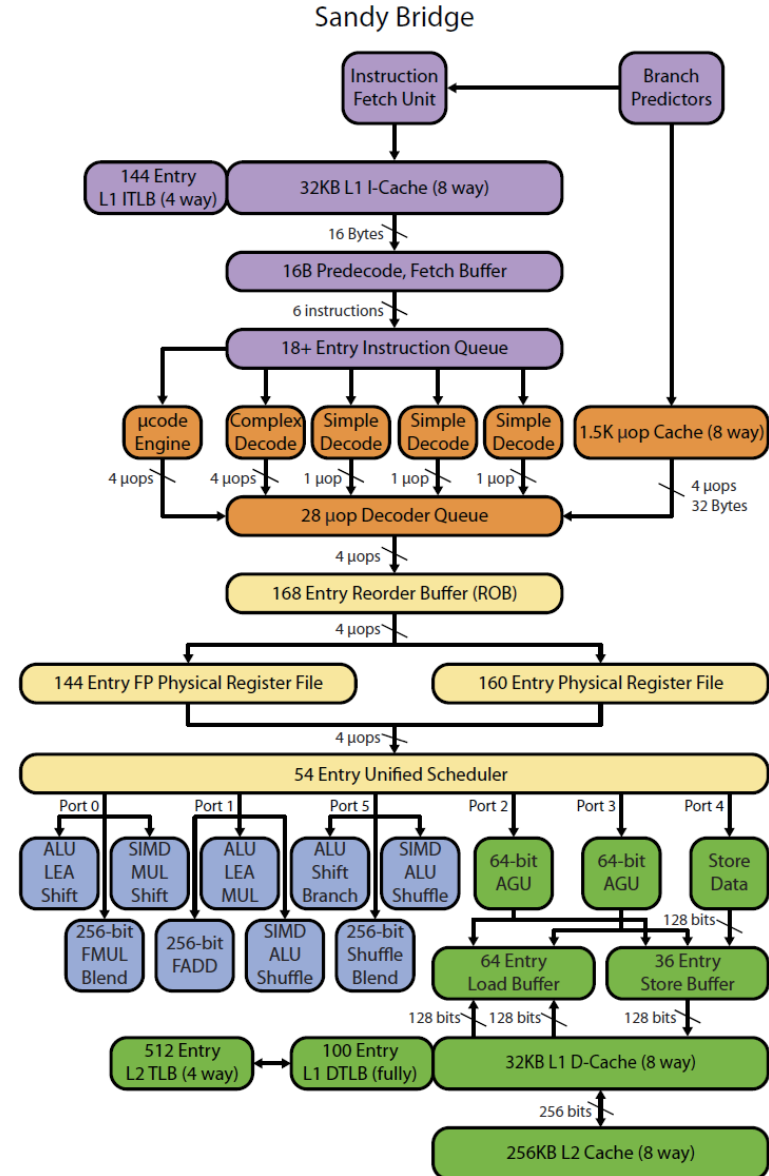
- A centralized bookkeeping table: tracks instructions + registers they depend or modify
- Status of result registers (who is going to write to a given register)
- Status of the functional units



Unit	Latency
Integer	0
Memory	1
FP Add	2
FP Multiply	10
FP Div	40



Sandy Bridge



Loop Unrolling

- Common loop optimization strategy
- Duplicate the body of the loop

```
for (int i=0; i < n ; i++)  
    z[i] = x[i] + y[i];
```

```
for (int i=0; i < n ; i++4){  
    z[i+0] = x[i+0] + y[i+0];  
    z[i+1] = x[i+1] + y[i+1];  
    z[i+2] = x[i+2] + y[i+2];  
    z[i+3] = x[i+3] + y[i+3];  
}
```

- Plays well with instruction level parallelism
- Register utilization, instruction scheduling
- Not always advantageous. **Why?**