CMPE 110: Computer Architecture

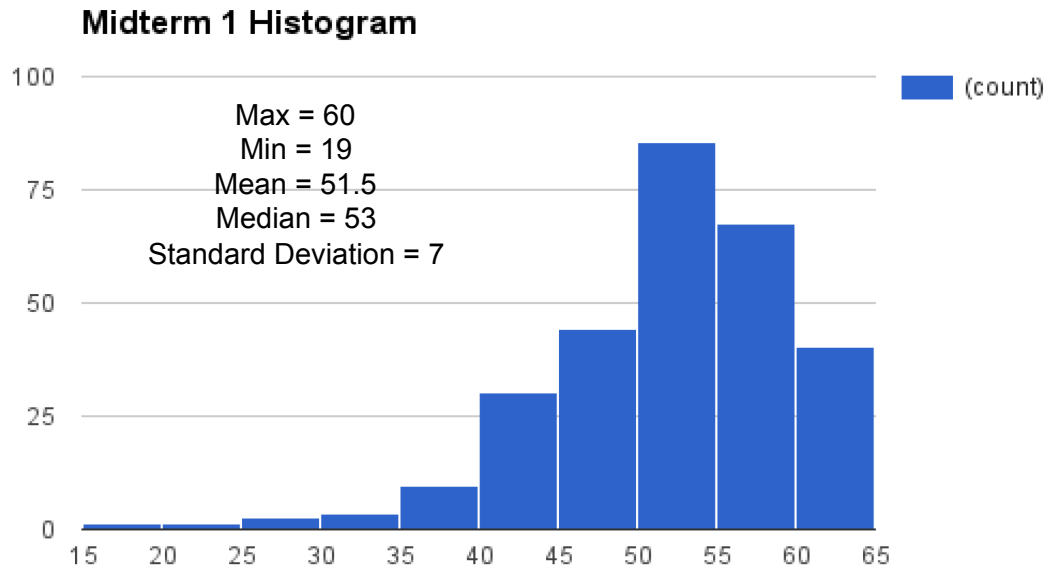
Week 5
Memory Hierarchy

Jishen Zhao (http://users.soe.ucsc.edu/~jzhao/)

[Adapted in part from Jose Renau, Mary Jane Irwin, Joe Devietti, Onur Mutlu, and others]

Reminder

- Midterm1 grade posted
  - Paper will be returned during TA office hours and discussion sessions (incl. students who are not enrolled in the class)



Review: Out-of-order execution

Basic idea of out-of-order execution:

- In-order F, D, W; out-of-order X (and M)
- Components added to pipeline:
  - Before X: reservation station
  - Before W: reorder buffer
- Register renaming
  - Eliminates false dependencies (those not true dependencies)

Must know

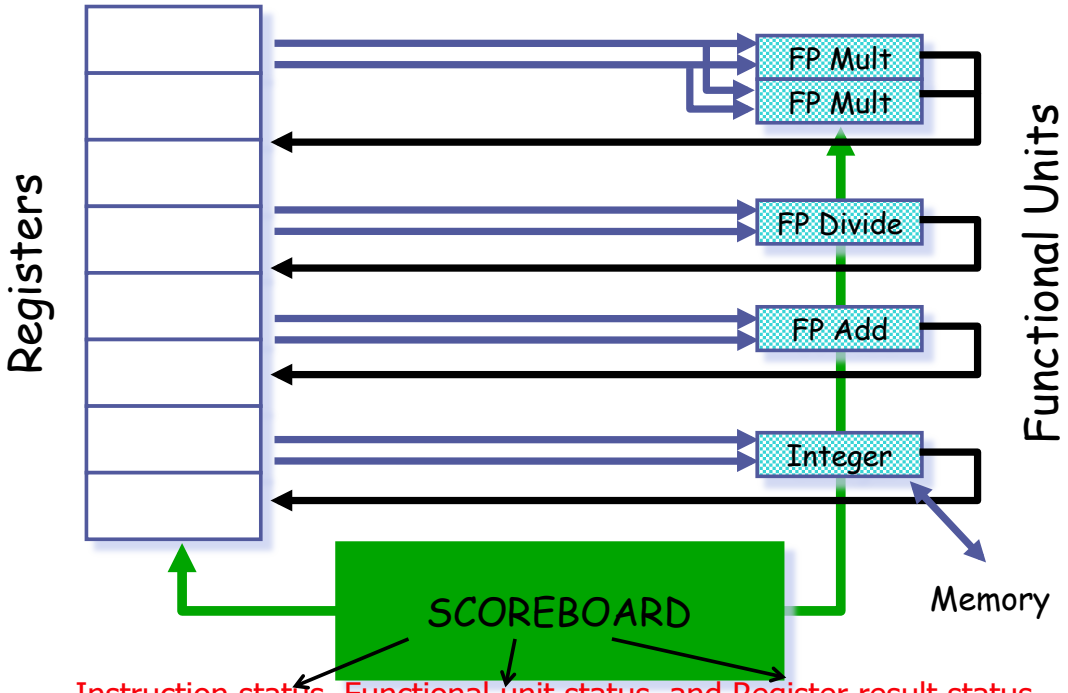
- How to draw pipeline table, when out-of-order execution is supported
- How to draw pipeline table, when register renaming is supported
- What are components added to pipeline to support out-of-order execution

Other than Tomasulo approach

"Scoreboarding" technique

Also a dynamic scheduling scheme

Scoreboard Architecture(CDC 6600 1963)



Instruction status, Functional unit status, and Register result status

(Which pipeline stage?) (Which being used by which inst?)

Scoreboarding basic idea

- "Out-of-order execution D stage jobs:
  1. Issue—decode instructions, check for structural (e.g., WAW) hazards
  2. Read operands—wait until no data hazards, then read operands
- Scoreboards date to CDC6600 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!
  - Imprecise interrupt/exception model for now

Four Stages of Scoreboard Control

- Handles all RAW, WAR, and WAW with proper stalls, but allows independent instructions to proceed
  - R = read, W = write, A = after

Data Dependence Types

Flow dependence

$r_3 \leftarrow r_1$ op r_2 — Read-after-Write

$r_5 \leftarrow r_3$ op r_4 — (RAW)

Anti dependence

$r_3 \leftarrow r_1$ op r_2 — Write-after-Read

$r_1 \leftarrow r_4$ op r_5 — (WAR)

Output-dependence

$r_3 \leftarrow r_1$ op r_2 — Write-after-Write

$r_5 \leftarrow r_3$ op r_4 — (WAW)

$r_3 \leftarrow r_6$ op r_7

Four Stages of Scoreboard Control

- After fetching (F) stage…
- Step1: Issue—decode instructions & check for structural hazards (D1 stage)
  - Instructions issued in program order (for hazard checking)
  - Don't issue if structural hazard
  - Don't issue if instruction is output dependent on any previously issued but uncompleted instruction (no **WAW** hazards)
- Step 2: Read operands—wait until no data hazards, then read operands (D2 stage)
  - All real dependencies (**RAW** hazards) resolved in this stage, since we wait for instructions to write back data.
  - No forwarding of data in this model!

Approaches to Dependence Detection

- Each register in register file has a Valid bit associated with it
- An instruction that is writing to the register resets the Valid bit
- An instruction in **Decode** stage checks if all its source and destination registers are Valid
  - Yes: No need to stall… No dependence
  - No: Stall the instruction

- Advantage:
  - Simple. 1 bit per register

- Disadvantage:
  - Need to stall for all types of dependences, not only flow dependency

Four Stages of Scoreboard Control

- Step 3: Execution—operate on operands (X)
  - The functional unit begins execution upon receiving operands. When the result is ready, it notifies the scoreboard that it has completed execution.
- Step 4: Write result—finish execution (W)
  - 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
  - **A more detailed example will be posted after class**

Scoreboard Implications

- Out-of-order completion => WAR, WAW data hazards?
- Solutions for WAR:
  - Stall writeback until registers have been read
  - Read registers only during Read Operands stage
- Solution for WAW:
  - Detect hazard and stall issue of new instruction until other instruction completes
- No register renaming!
- Need to have multiple instructions in execution phase => multiple execution units or pipelined execution units
- Scoreboard keeps track of dependencies between instructions that have already issued.
- Scoreboard replaces D, X, W with 4 stages

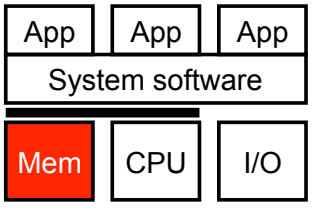
Key idea of Scoreboard

- Allow instructions behind stall to proceed
  - Extend D stage to -- Issue instr & read operands
  - D stage checked both for structural & data dependencies
  - Enables out-of-order execution and out-of-order completion
- Original version (CDC 6600) did not handle forwarding
- No automatic register renaming

Can we do better?

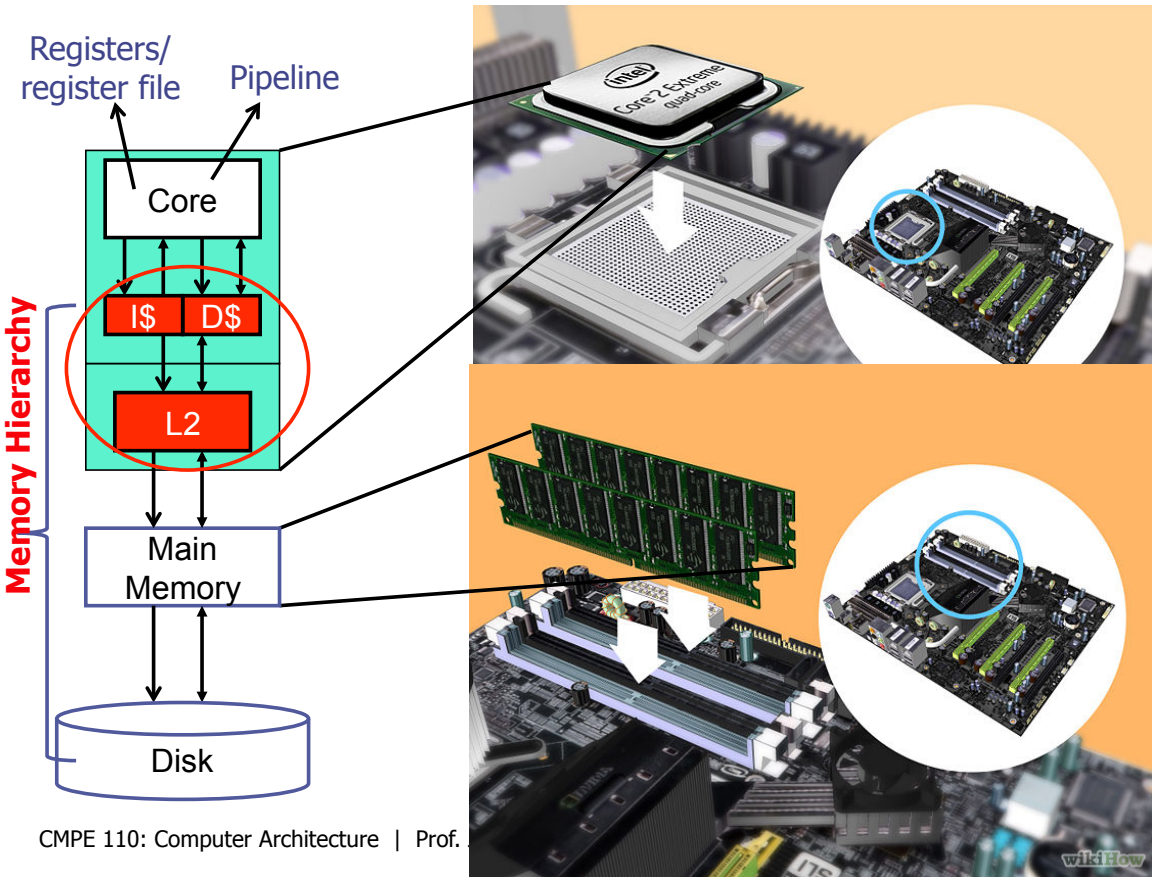
- Not Stalling on Anti and Output Dependences
- What changes would you make to the scoreboard to enable this?

Memory Hierarchy



- Introduction to memory hierarchy
  - What is memory hierarchy
  - Why memory hierarchy
- Basic idea of caching
- Basic idea of main memory (hardware design)

Memory hierarchy: what, where, why

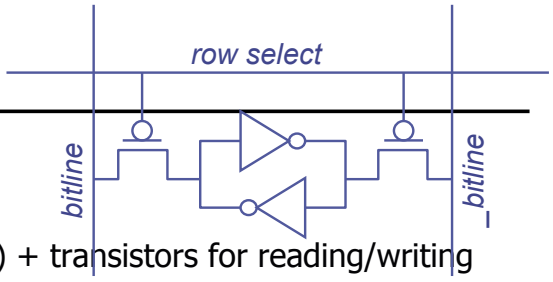


A story of the memory hierarchy

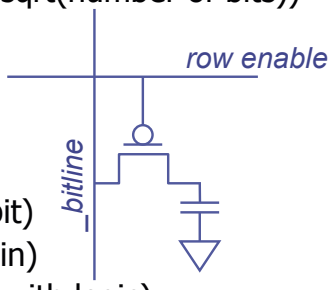
- Processor can compute only as fast as memory
  - A 3Ghz processor can execute an "add" operation in 0.33ns
  - Today's "main memory" latency is more than 33ns
  - Naïve implementation:
    - loads/stores can be 100x slower than other operations
- Unobtainable goal with one level of main memory:
  - Memory that operates at processor speeds
  - Memory as large as needed for all running programs
  - Memory that is cost effective

- Can't find a technology that achieves all of these goals
  - E.g., SRAM: fast, low density, expensive
    DRAM: slower, high density, less expensive

Types of Memory



- **Static RAM (SRAM)**
  - 6 or 8 transistors per bit
    - Two inverters (4 transistors) + transistors for reading/writing
  - Optimized for speed
  - **Fast** (sub-nanosecond latencies for small SRAM)
    - Speed roughly proportional to its area (~ sqrt(number of bits))
  - Mixes well with standard processor logic



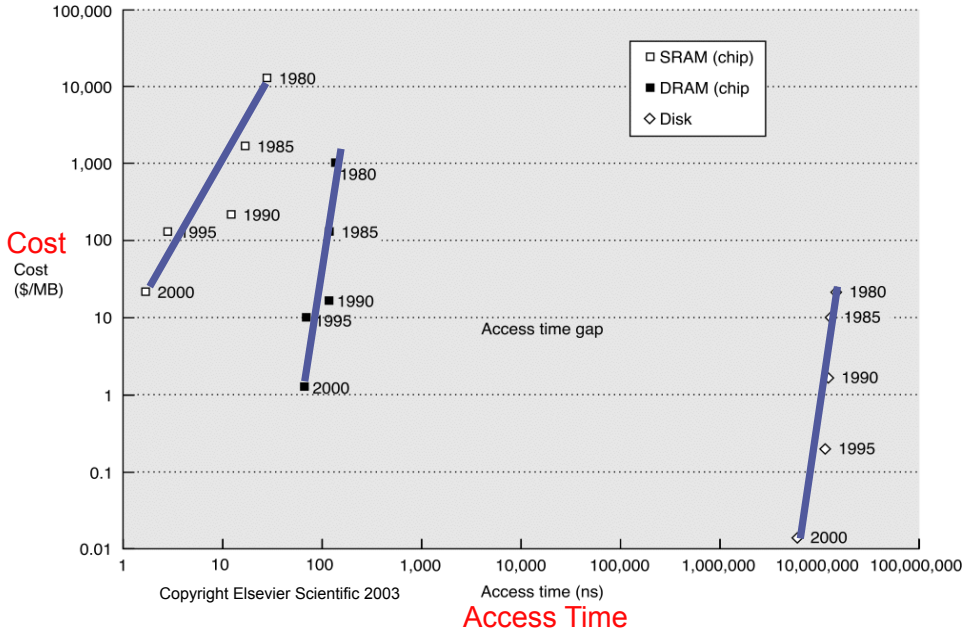
- **Dynamic RAM (DRAM)**
  - 1 transistor + 1 capacitor per bit
  - Optimized for **density** (in terms of cost per bit)
  - **Slow** (>30ns internal access, ~50ns pin-to-pin)
  - Different fabrication steps (does not mix well with logic)

- **Nonvolatile storage:** Magnetic disk, flash memory, etc.

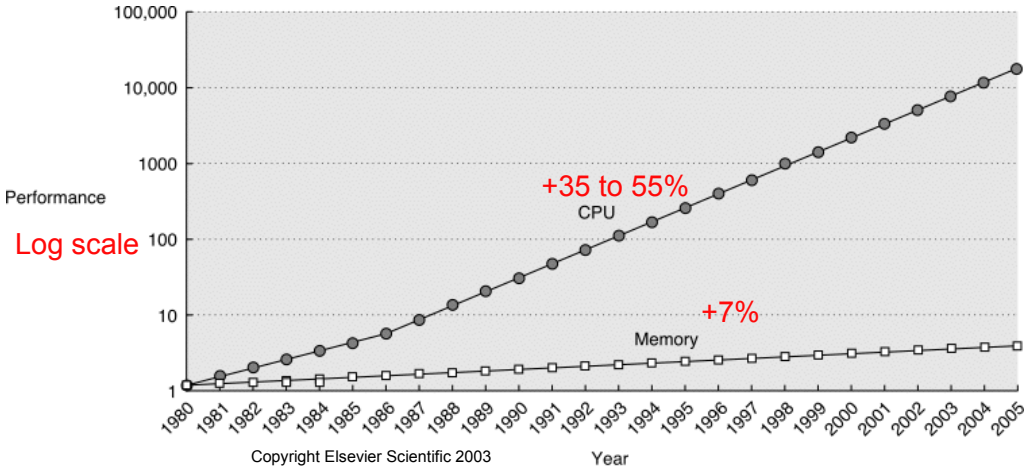
Memory & Storage Technologies

- **Cost** - what can $200 buy (2009)?
  - SRAM: 16MB
  - DRAM: 4,000MB (4GB) – 250x cheaper than SRAM
  - Flash: 64,000MB (64GB) – 16x cheaper than DRAM
  - Disk: 2,000,000MB (2TB) – 32x vs. Flash (512x vs. DRAM)
- **Latency**
  - SRAM: <1 to 2ns (on chip)
  - DRAM: ~50ns – 100x or more slower than SRAM
  - Flash: 75,000ns (75 microseconds) – 1500x vs. DRAM
  - Disk: 10,000,000ns (10ms) – 133x vs Flash (200,000x vs DRAM)
- **Bandwidth**
  - SRAM: 300GB/sec (e.g., 12-port 8-byte register file @ 3Ghz)
  - DRAM: ~25GB/s
  - Flash: 0.25GB/s (250MB/s), 100x less than DRAM
  - Disk: 0.1 GB/s (100MB/s), 250x vs DRAM, **sequential** access only

Memory Technology Trends

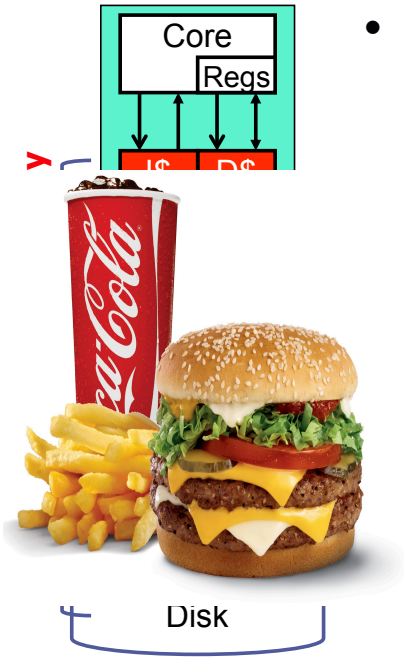


The "Memory Wall": if we only use DRAM



- Processors get faster more quickly than memory (note log scale)
  - Processor speed improvement: 35% to 55%
  - Memory latency improvement: 7%

Memory Hierarchy: what, where, why



- What are the components in a memory hierarchy?
  - Caches
  - Main memory
  - Storage devices (e.g., hard drives, SSD, etc.)
- Where do these components locate?
  - On processor chip
  - Off processor chip
- Why do we need a hierarchy of memory
  - Goals: High **speed**, large **capacity**, low **cost**
  - No single memory technology can simultaneously achieve all three goals
  - Solution: build a Combo

A Closer Look at The Memory Hierarchy

Known From the Beginning

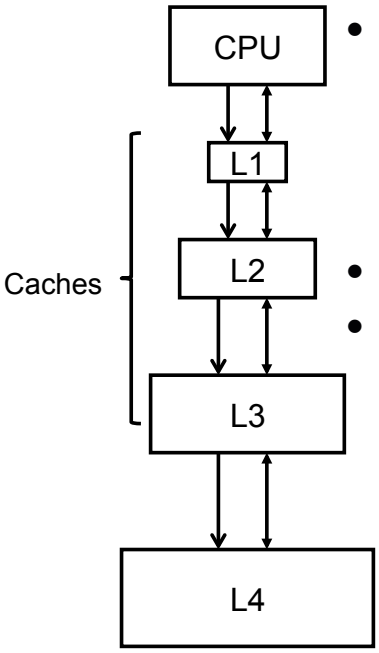
"Ideally, one would desire an infinitely large memory capacity such that any particular word would be immediately available … We are forced to recognize the possibility of constructing a hierarchy of memories, each of which has a greater capacity than the preceding but which is less quickly accessible."

Burks, Goldstine, VonNeumann

"Preliminary discussion of the logical design of an electronic computing instrument"

IAS memo 1946

Memory Hierarchy



- Hierarchy of memory components
  - Upper level components
    - Fast ↔ Small ↔ Expensive
  - Lower components
    - Slow ↔ Big ↔ Cheap
- Connected by "buses"
- Most frequently accessed data in L1
  - next most frequently accessed data in L2, etc.
  - Move data up-down hierarchy

Scoreboarding

Adapted from Mike Scott

A scoreboard is a hardware unit that takes full responsibility for instruction issue and execution, including all hazard detection. Every instruction goes through the scoreboard, where a record of data dependencies is constructed and maintained.

The scoreboard determines when an instruction can read its operands and begin execution. If an instruction cannot be executed immediately, it monitors the hardware resources and decides when an instruction can execute. It also determines when an instruction can write-back its results to a destination register.

An example:-

```
L.D     F6,34(R2)
L.D     F2,45(R3)
MUL.D   F0,F2,F4
SUB.D   F8,F6,F2
DIV.D   F10,F0,F6
ADD.D   F6,F8,F2
```

The scoreboard has 3 components; *Instruction status, Functional unit status,* and *Register result status*.

A snap-shot of the scoreboard just after starting execution:-

Instruction	*Issue*	*Read Operands*	*Ex*	*WB*
L.D F6,34(R2)	Yes	Yes	Yes	Yes
L.D F2,45(R3)	Yes	Yes	Yes	
MUL.D F0,F2,F4	Yes			
SUB.D F8,F6,F2	Yes			
DIV.D F10,F0,F6	Yes			

```
ADD.D F6,F8,F2
```

Unit	*Busy*	*Op*	*Fi*	*Fj*	*Fk*	*Qj*	*Qk*	*Rj*	*Rk*
Int	Yes	Load	F2	R3				No	
Mult1	Yes	Mult	F0	F2	F4	Int		No	Yes
Add	Yes	Sub	F8	F6	F2		Int	Yes	No
Divide	Yes	Div	F10	F0	F6	Mult		No	Yes

Register results

F0	F2	F4	F6	F8	F10..
Mult1	Int			Add	Divide

Fi is the destination register, Fj and Fk the two source registers. Qj and Qk indicate which functional unit will produce this operand. Rj and Rk are flags which indicate if the associated register is ready but not yet read.

At this stage the first L.D has completed, and the second is just about to write its result. The MUL.D, SUB.D and DIV.D are all issued, but stalled, waiting for their operands. For example the multiply unit is waiting for the integer unit (to supply F2), the add/subtract unit is also waiting for F2, and the divide unit is waiting for the Multiply unit.

There are RAW hazards between the second L.D and the MUL.D and SUB.D instructions., from MUL.D to DIV.D, and from SUB.D to ADD.D. There is a WAR hazard between DIV.D and ADD.D

The ADD.D instruction cannot issue – it needs the same functional unit that is currently is use by the SUB.D instruction. This is a structural hazard/stall.

Now lets fast forward a few cycles to when the MUL.D is ready to write its results. Assume that the Add unit takes 2 clock cycles, the Multiply unit takes 10 clock cycles, and the Divide takes 40 clock cycles..

Instruction	*Issue*	*Read Operands*	*Ex*	*WB*
L.D F6,34(R2)	Yes	Yes	Yes	Yes
L.D F2,45(R3)	Yes	Yes	Yes	Yes
MUL.D F0,F2,F4	Yes	Yes	Yes	
SUB.D F8,F6,F2	Yes	Yes	Yes	Yes
DIV.D F10,F0,F6	Yes			
ADD.D F6,F8,F2	Yes	Yes	Yes	

Unit	*Busy*	*Op*	*Fi*	*Fj*	*Fk*	*Qj*	*Qk*	*Rj*	*Rk*
Int	No								
Mult1	Yes	Mult	F0	F2	F4			No	No
Add	Yes	Add	F6	F8	F2			No	No
Divide	Yes	Div	F10	F0	F6	Mult		No	Yes

Register results

F0	F2	F4	F6	F8	F10..
Mult1			Add		Divide

The DIV.D instruction is still waiting for F0 to become available. The ADD.D has kicked off as the SUB.D has now completed, and the addition functional unit is now free. However ADD.D cannot complete yet due to the WAR hazard on F6.

Once DIV.D gets its F6 operand, ADD.D is able to complete. DIV.D completes last.

As we have seen by using register renaming WAR and WAW hazards can be avoided altogether.