Complex Pipelining

Krste Asanovic
Laboratory for Computer Science
M.I.T.

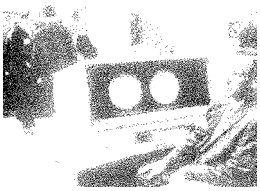
http://www.csg.lcs.mit.edu/6.823

Motivation

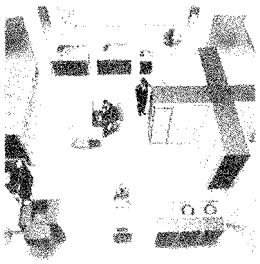
Pipelining becomes complex when we want high performance in the presence of

- **Memory systems with variable access time**
- **Long latency or partially pipelined floating point unit**
- **Multiple function and memory units**

CDC 6600 *Seymour Cray, 1963*



- A fast *pipelined machine* with 60-bit words
  (128 Kword main memory capacity)
- *Ten functional units (parallel, unpipelined)*
  Floating Point *adder, 2 multipliers, divider*
  Integer *adder, 2 incrementers, ...*
- Hardwired control *(no microcoding)*
- *Dynamic scheduling* of instructions using a *scoreboard*



- Ten *Peripheral Processors* for Input/Output
  - a fast *multi-threaded* 12-bit integer ALU
- Very fast clock, 10 MHz, >400,000 transistors, 750 sq. ft., 12,000 lbs, 150 kW, novel *freon-based* technology for cooling
- Fastest machine in world for 5 years (until 7600)
  – over 100 sold ($7-10M each)

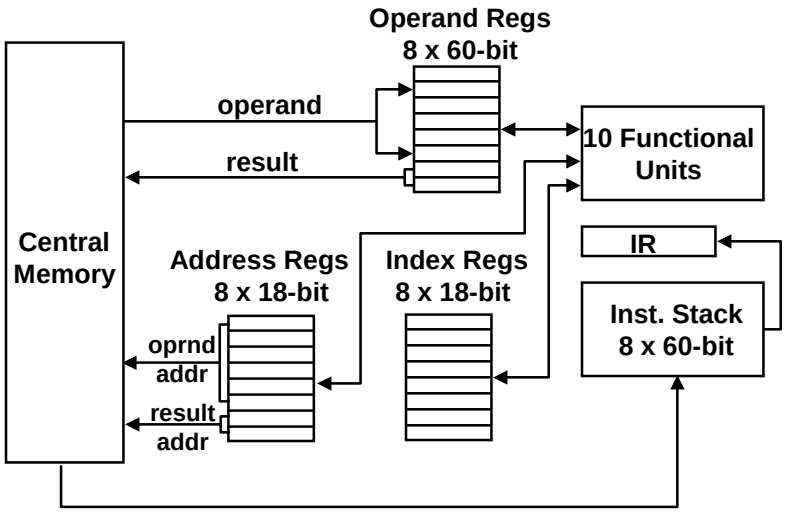
IBM Memo on CDC6600

Thomas Watson Jr., IBM CEO, August 1963:

> "Last week, Control Data ... announced the 6600 system. I understand that in the laboratory developing the system there are only 34 people including the janitor. Of these, 14 are engineers and 4 are programmers... Contrasting this modest effort with our vast development activities, I fail to understand why we have lost our industry leadership position by letting someone else offer the world's most powerful computer."

To which Cray replied: "It seems like Mr. Watson has answered his own question."

CDC 6600: Datapath



CDC 6600: A Load/Store Architecture

- ***Separate instructions to manipulate three types of reg.***
  - **8 60-bit data registers (X)**
  - **8 18-bit address registers (A)**
  - **8 18-bit index registers (B)**
- ***All arithmetic and logic instructions are reg-to-reg***

6	3	3	3
opcode	i	j	k

Ri ← (Rj) op (Rk)

- ***Only Load and Store instructions refer to memory!***

6	3	3	18
opcode	i	j	disp

Ri ← M[(Rj) + disp]

- ***Touching address registers 1 to 5 initiates a load***
  6 to 7 initiates a store
  - very useful for vector operations

CDC6600: Vector Addition

```
        B0 ← - n
loop:   JZE  B0, exit
        A0 ← B0 + a0      load X0
        A1 ← B0 + b0      load X1
        X6 ← X0 + X1
        A6 ← B0 + c0      store X6
        B0 ← B0 + 1
        jump loop
```


Realistic Memory Systems

Latency of access to the main memory is usually much greater than one cycle and often unpredictable

Solving this problem is a central issue in computer architecture

Common solutions

- **separate instruction and data memory ports and buffers**
  - ⇒ ***no self-modifying code***
- **caches**
  - ***single cycle except in case of a miss ⇒ stall***
- **interleaved memory**
  - ***multiple memory accesses ⇒ stride conflicts***
- **split-phase memory operations**
  - ⇒ ***out-of-order responses***

Floating Point Unit

Much more hardware than an integer unit

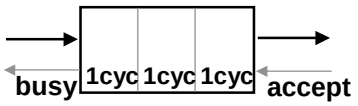
Single-cycle floating point units are impractical

- **it is common to have several floating point units**
- **it is common to have different types of FPU's**
  Fadd, Fmul, Fdiv, ...
- **an FPU may be pipelined, partially pipelined or not pipelined**

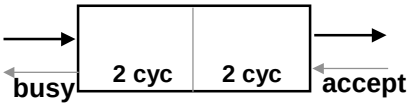
To operate several FPU's concurrently the register file needs to have more read and write ports

Function Unit Characteristics

fully pipelined



partially pipelined

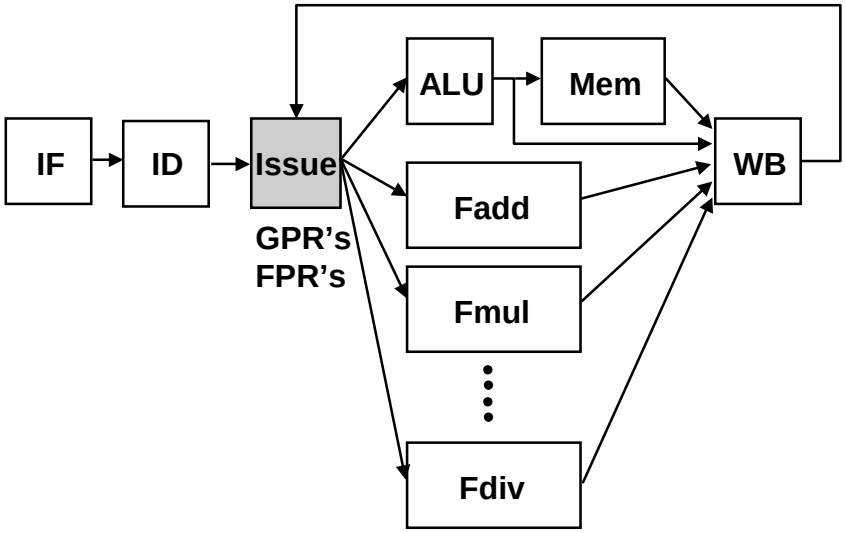


Function units have internal pipeline registers

⇒ **operands are latched when an instruction enters a function unit**

⇒ **inputs to a function unit (e.g., register file) can change during a long latency operation**

Multiple Function Units



Floating Point ISA

Interaction between the Floating point datapath and the Integer datapath is determined largely by the ISA

DLX ISA

- **separate register files for FP and Integer instructions**
- **separate load/store for FPR's and GPR's but both use GPR's for address calculation**
- **separate conditions for branches**
  - **FP branches are defined in terms of condition codes**
- ***the only interaction is via a set of move instructions (some ISA's don't even permit this)***

New Possibilities of Hazards

- structural conflicts at the write-back stage due to variable latencies of different function units
- structural conflicts at the execution stage if some FPU or memory unit is not pipelined and takes more than one cycle
- out-of-order write hazards due to variable latencies of different function units

appropriate pipeline interlocks can resolve all these hazards

Write-Back Stage Arbitration

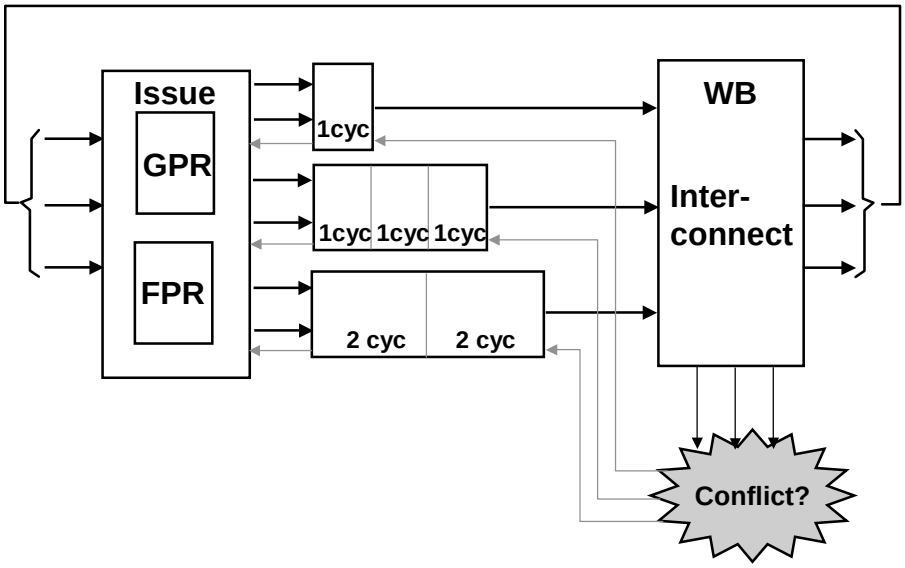
Is the latency of each function unit fixed and known?

Yes - structural hazards can be avoided by delaying the instruction from entering the execution phase

No - WB stage has to be arbitrated dynamically
- ⇒ WB stage may exert back pressure on a function unit
- ⇒ Issue may not dispatch an instruction into that unit until the unit is free

We will assume that the latencies are not known
(This is the more general case)

WB and Back Pressure



Dependence Analysis

Types of Data Hazards

Consider executing a sequence of

$$r_k \leftarrow (r_i) \text{ op } (r_j)$$

type of instructions

Data-dependence

$r_3 \leftarrow (r_1)$ op (r_2) — **Read-after-Write**
$r_5 \leftarrow (r_3)$ op (r_4) — **(RAW) hazard**

Anti-dependence

$r_3 \leftarrow (r_1)$ op (r_2) — **Write-after-Read**
$r_1 \leftarrow (r_4)$ op (r_5) — **(WAR) hazard**

Output-dependence

$r_3 \leftarrow (r_1)$ op (r_2) — **Write-after-Write**
$r_5 \leftarrow (r_3)$ op (r_4) — **(WAW) hazard**
$r_3 \leftarrow (r_6)$ op (r_7)

Detecting Data Hazards

Range and Domain of instruction i

R(i) = Registers (or other storage) modified by instruction i
D(i) = Registers (or other storage) read by instruction i

Suppose instruction j follows instruction i in the program order. Executing instruction j before the effect of instruction i has taken place can cause a

RAW hazard if $R(i) \cap D(j) \neq \emptyset$

WAR hazard if $D(i) \cap R(j) \neq \emptyset$

WAW hazard if $R(i) \cap R(j) \neq \emptyset$

Register vs. Memory Data Dependence

Data hazards due to register operands can be determined at the decode stage

but

data hazards due to memory operands can be determined only after computing the effective address

store	$M[(r_1) + disp1] \leftarrow (r_2)$
load	$r_3 \leftarrow M[(r_4) + disp2]$

Data Hazards: An Example

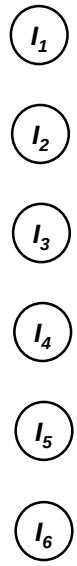
worksheet

I_1	DIVD	f6,	f6,	f4
I_2	LD	f2,	45(r3)	
I_3	MULTD	f0,	f2,	f4
I_4	DIVD	f8,	f6,	f2
I_5	SUBD	f10,	f0,	f6
I_6	ADDD	f6,	f8,	f2

Instruction Scheduling

I_1	DIVD	f6,	f6,	f4
I_2	LD	f2,	45(r3)	
I_3	MULTD	f0,	f2,	f4
I_4	DIVD	f8,	f6,	f2
I_5	SUBD	f10,	f0,	f6
I_6	ADDD	f6,	f8,	f2

in-order	I_1	I_2	I_3	I_4	I_5	I_6
out-of-order	I_2	I_1	I_3	I_4	I_5	I_6
out-of-order	I_1	I_2				



Out-of-order Completion

In-order Issue

					Latency
I_1	DIVD	f6,	f6,	f4	*4*
I_2	LD	f2,	45(r3)		*1*
I_3	MULTD	f0,	f2,	f4	*3*
I_4	DIVD	f8,	f6,	f2	*4*
I_5	SUBD	f10,	f0,	f6	*1*
I_6	ADDD	f6,	f8,	f2	*1*

in-order comp	1	2			<u>1</u>	<u>2</u>	3	4		<u>3</u>	5	<u>4</u>	6	<u>5</u>	<u>6</u>
out-of-order comp	1	2	<u>2</u>	3	<u>1</u>	4	<u>3</u>	5	<u>5</u>	<u>4</u>	6	<u>6</u>			

Scoreboard:
A Data Structure to Detect Hazards

When is it Safe to Issue an Instruction?

For each instruction at the Issue stage the following checks need to be made

- **Is the required function unit available?**
- **Is the input data available? ⇒ RAW?**
- **Is it safe to write the destination? ⇒ WAR?**
  WAW?
- **Is there a structural conflict at the WB stage?**

Assume there is only one instruction in the Issue stage and if it cannot be issued then the Instruction Fetch and Decode stall

A Data Structure for Correct Issues

Keeps track of the status of Functional Units

name	*busy*	*op*	*dest*	*src1*	*src2*
Int					
Mem					
Add1					
Add2					
Add3					
Mult1					
Mult2					
Div					

The instruction i at the Issue stage consults this table

FU available?	check the busy column
RAW?	search the dest column for i's sources
WAR?	search the source columns for i's destination
WAW?	search the dest column for i's destination

An entry is added to the table if no hazard is detected;
An entry is removed from the table after Write-Back

Simplifying the Data Structure Assuming In-order Issue

Suppose the instruction is not dispatched by the Issue stage if a RAW hazard exists or the required FU is busy

Can the dispatched instruction cause a

WAR hazard ?
why?

WAW hazard ?

Simplifying the Data Structure ...

No WAR hazard ⇒ no need to keep *src1* and *src2*

The Issue stage does not dispatch an instruction in case of a WAW hazard

⇒ a register name can occur at most once in the *dest* column

WP[reg#] : a bit-vector to record the registers for which writes are pending

These bits are set to true by the Issue stage and set to false by the WB stage

⇒ Each pipeline stage in the FU's must carry the *dest* field and a flag to indicate if it is valid

the (we, ws) pair

Scoreboard for In-order Issues

Busy[unit#] : a bit-vector to indicate unit's availability. (unit = Int, Add, Mult, Div)
These bits are hardwired to FU's.

WP[reg#] : a bit-vector to record the registers for which writes are pending

Issue checks the instruction (opcode dest src1 src2) against the scoreboard (Busy & WP) to dispatch

FU available?	***not* Busy[FU#]**
RAW?	**WP[src1] or WP[src2]**
WAR?	***cannot arise***
WAW?	**WP[dest]**

Scoreboard Dynamics *worksheet*

		Functional Unit Status										Registers Reserved for Writes
		Int(1)	Add(1)	Mult(3)			Div(4)				WB	
I_1	t0						f6					f6
I_2	t1	f2						f6				f6, f2
	t2											
	t3											
	t4											
	t5											
	t6											
	t7											
	t8											
	t9											
	t10											
	t11											

I_1	DIVD	f6,	f6,	f4
I_2	LD	f2,	45(r3)	
I_3	MULTD	f0,	f2,	f4
I_4	DIVD	f8,	f6,	f2
I_5	SUBD	f10,	f0,	f6
I_6	ADDD	f6,	f8,	f2

FU available?
RAW?
WAW?

Scoreboard Dynamics

		Functional Unit Status										Registers Reserved for Writes	
		Int(1)	Add(1)	Mult(3)			Div(4)				WB		
I_1	t0						f6					f6	
I_2	t1	f2						f6				f6, f2	
	t2								f6		f2	f6, f2	I_2
I_3	t3			f0						f6		f6, f0	
	t4				f0						f6	f6, f0	I_1
I_4	t5					f0	f8					f0, f8	
	t6							f8			f0	f0, f8	I_3
I_5	t7		f10						f8			f8, f10	
	t8									f8	f10	f8, f10	I_5
	t9										f8	f8	I_4
I_6	t10		f6									f6	
	t11										f6	f6	I_6

I_1	DIVD	f6,	f6,	f4
I_2	LD	f2,	45(r3)	
I_3	MULTD	f0,	f2,	f4
I_4	DIVD	f8,	f6,	f2
I_5	SUBD	f10,	f0,	f6
I_6	ADDD	f6,	f8,	f2