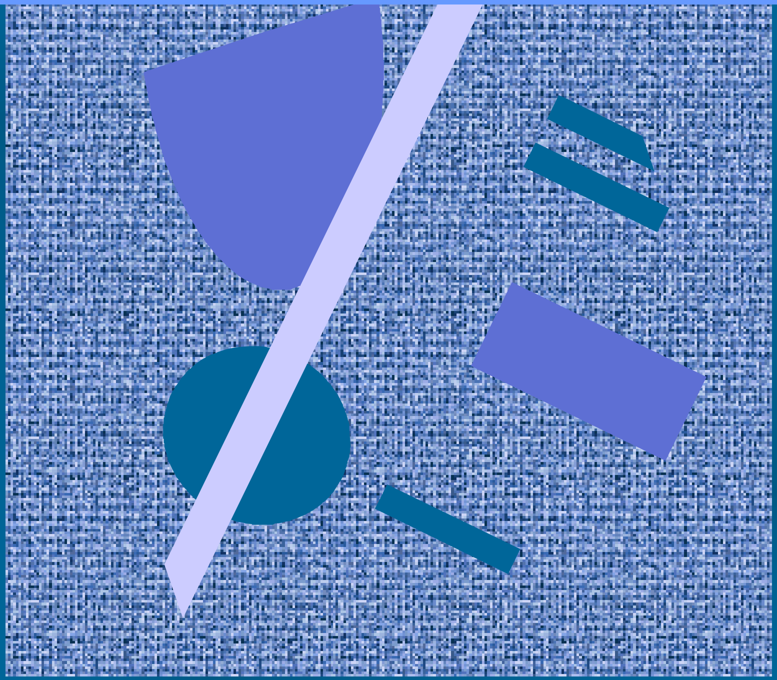
CPU Structure and Function Ch 11



General Organisation

Registers

Instruction Cycle

Pipelining

Branch Prediction

Interrupts

General CPU Organization (4)

- ALU
  - does all <u>real</u> work
- Registers
  - data stored here
- Internal CPU Bus
- Control
  - determines who does what when
  - driven by clock
  - uses control signals (wires) to control what every circuit is doing at any given clock cycle

Fig. 11.1

Fig. 11.2

More in Chapters 14-15

Register Organisation (2)

- Registers make up CPU work space
  - User visible registers
    - accessible directly via instructions
  - Control and status registers
    - may be accessible indirectly via instructions
    - may be accessible only internally
- Internal latches for temporary storage during instruction execution
  - E.g., ALU operand either from constant in instruction or from machine register

ADD R1,R2,R3

BNeq Loop

HW exception

User Visible Registers

- Varies from one architecture to another
- General purpose
  - Data, address, index, PC, condition, ….
- Data
  - Int, FP, Double, Index
- Address
- Segment and stack pointers
  - only privileged instruction can write?
- Condition codes
  - result of some previous ALU operation

Control and Status Registers (5)

- PC
  - next instruction (<u>not</u> current!)
  - part of process state
- IR, Instruction (Decoding) Register
  - current instruction
- MAR, Memory Address Register
  - current memory address
- MBR, Memory Buffer Register
  - current data to/from memory
- PSW, Program Status Word
  - what is allowed? What is going on?
  - part of process state

Fig. 11.7

PSW - Program Status Word (8)

- Sign, zero?
- Carry (for multiword ALU ops)?
- Overflow?
- Interrupts that are enabled/disabled?
- Pending interrupts?
- Cpu execution mode (supervisor, user)?
- Stack pointer, page table pointer?
- I/O registers?

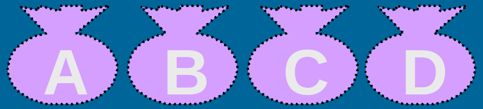
Instruction Cycle

- Basic cycle with interrupt handling Fig. 11.4
- Indirect cycle Figs 11.5-6
- Data Flow
  - – CPU, Bus, Memory Figs 11.7-9
- Data Path Fig 14.5
  - – inside CPU

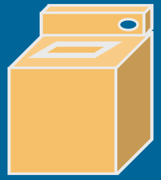
Pipeline Example

(liukuhihna)

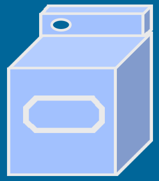
- Laundry Example (David A. Patterson)
- Ann, Brian, Cathy, Dave
  each have one load of clothes
  to wash, dry, and fold



- Washer takes 30 minutes

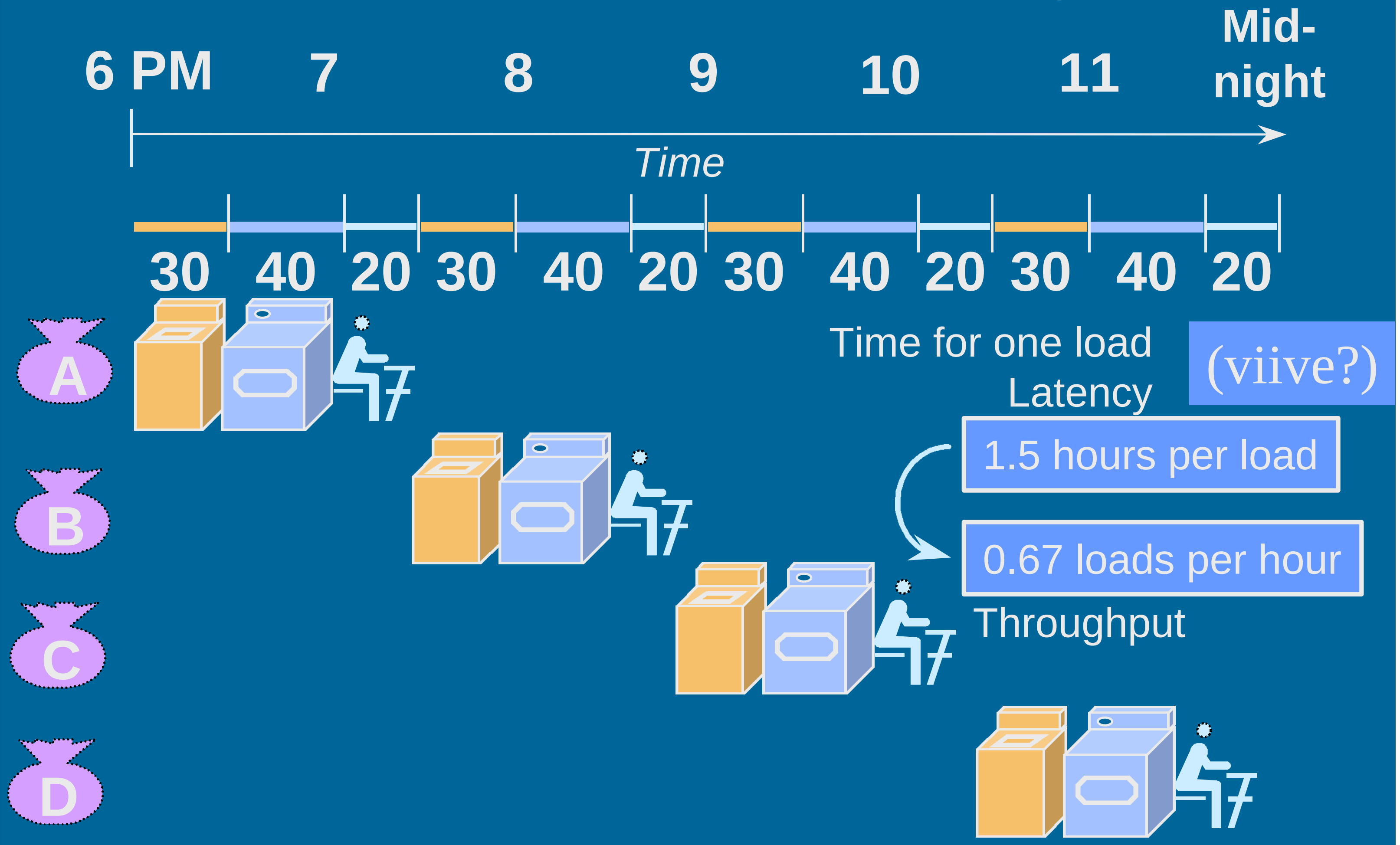


- Dryer takes 40 minutes



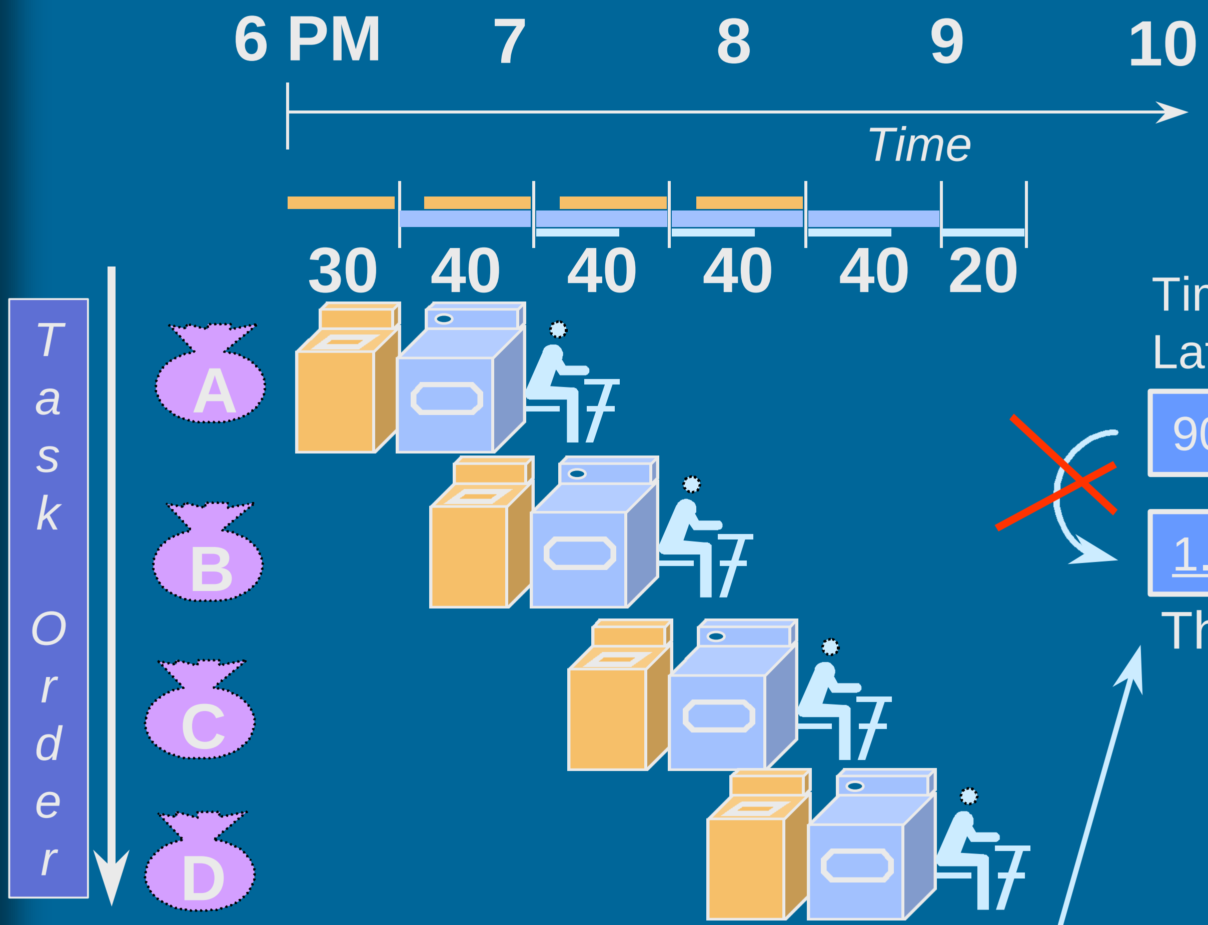
- “Folder” takes 20 minutes

Sequential Laundry (6)



- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

Pipelined Laundry (8)



Time for <u>one load</u>

Latency

90 minutes per load

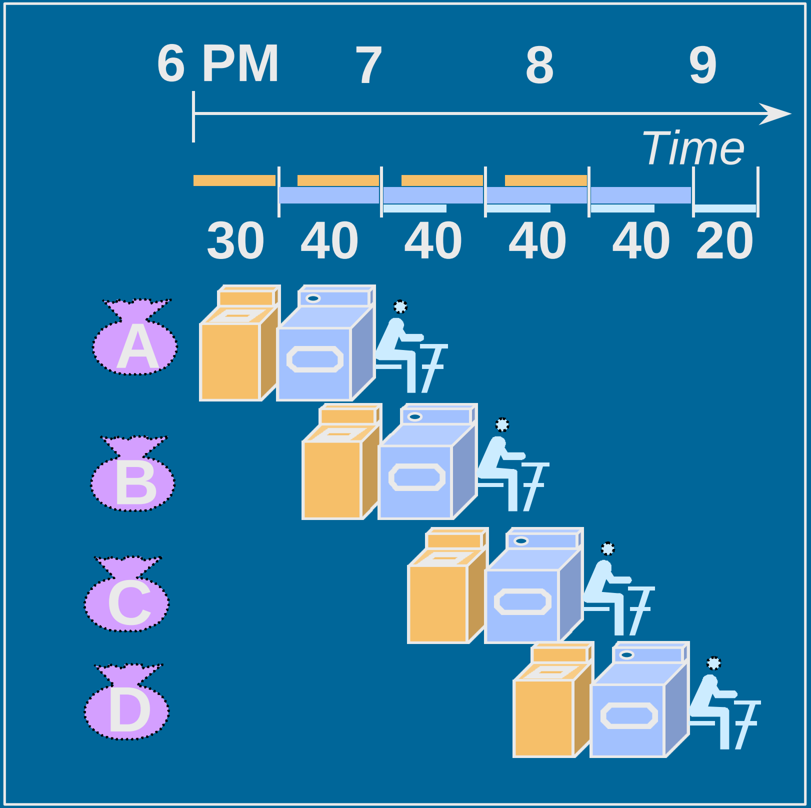
<u>1.15</u> loads per hour

Throughput

- Pipelined laundry takes <u>3.5 hours for 4 loads</u>

Pipelining Lessons (4)

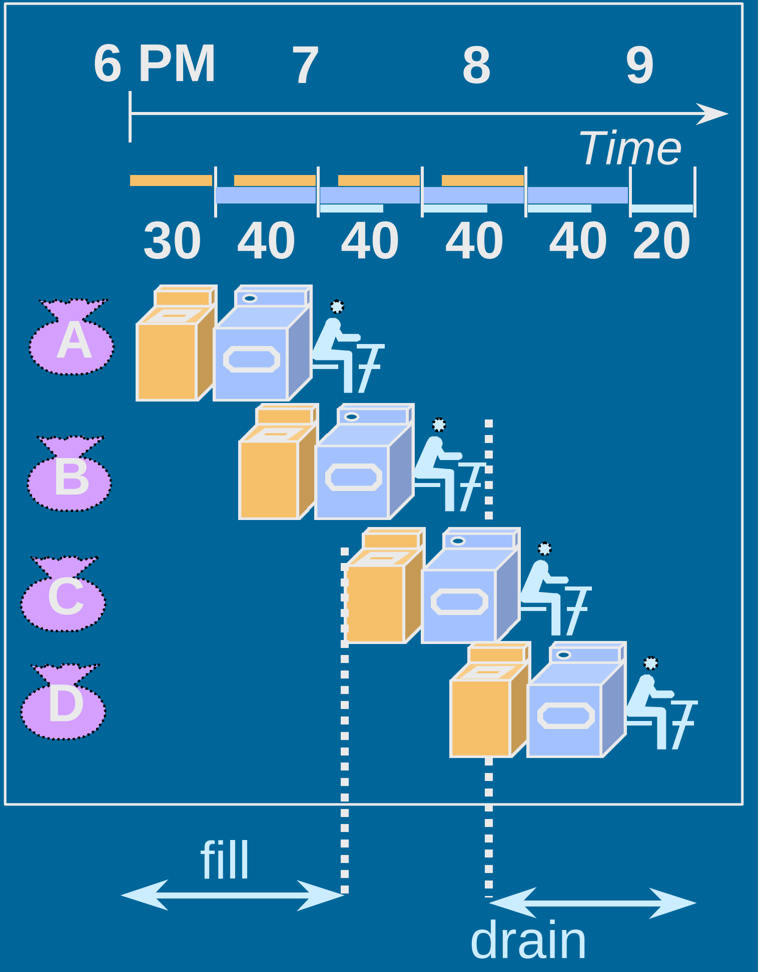
- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously
- Potential speedup = Number pipe stages



(nopeutus)

Pipelining Lessons (3)

- Unbalanced lengths of pipe stages reduces speedup
- May need more resources
  - Enough electrical current to run both washer and dryer simultaneously?
  - Need to have at least 2 people present all the time?
- Time to "fill" pipeline and time to "drain" it reduces speedup



2-stage Instruction Execution Pipeline

Fig. 11.10

- Good: instruction pre-fetch at the same time as execution of previous instruction
- Bad: execution time is longer, I.e., fetch stage is sometimes idle
- Bad: Sometimes (jump, branch) wrong instruction is fetched
  - every 6th instruction?
- Not enough parallelism $\Rightarrow$ more stages?

Another Possible Instruction Execution Pipeline

- FE - Fetch instruction
- DI - Decode instruction
- CO - Calculate operand effective addresses
- FO - Fetch operands from memory
- EI - Execute Instruction
- WO - Write operand (result) to memory

Pipeline Speedup (3)

No pipeline, 9 instructions $\xrightarrow{9 * 6}$ 54 time units

6 stage pipeline, 9 instructions $\xrightarrow{\text{Fig. 11.11}}$ 14 time units

$$\text{Speedup} = \frac{\text{Time}_{\text{old}}}{\text{Time}_{\text{new}}} = 54/14 = 3.86 \;< \mathbf{6\,!}$$

(nopeutus)

- Not every instruction uses every stage
  - – serial execution actually even faster
  - – speedup even smaller
  - – will not affect pipeline speed
  - – unused stage $\Rightarrow$ CPU idle (execution “bubble”)

Pipeline Execution Time (3)

- Time to execute one instruction (latency, seconds) may be longer than for non-pipelined machine
  - extra latches to store intermediate results
- Time to execute 1000 instructions (seconds) is shorter than that for non-pipelined machine,
  I.e.,
  Throughput (instructions per second) for pipelined machine is better (bigger) than that for non-pipelined machine
- Is this good or bad? Why?

Pipeline Speedup Problems

- Some stages are shorter than the others
- Dependencies between instructions
  - – Control dependency
    - E.g., conditional branch decision know only after EI stage

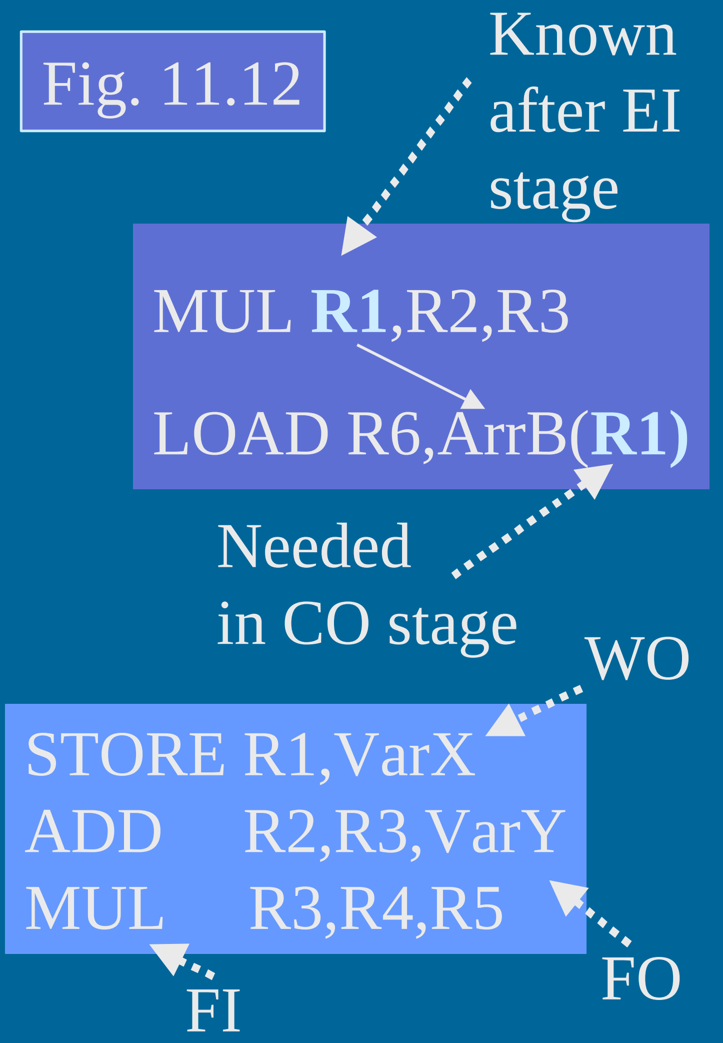
Fig. 11.12

Fig. 11.13

Pipeline Speedup Problems

- Dependencies between instructions
  - data dependency
    - E.g., one instruction depends on <u>some earlier</u> instruction
  - structural dependency
    - E.g., many instructions need the same resource <u>at the same time</u>
      - e.g., memory bus

Fig. 11.12



```
MUL **R1**,R2,R3
LOAD R6,ArrB(**R1**)
```

```
STORE R1,VarX
ADD    R2,R3,VarY
MUL    R3,R4,R5
```

Cycle Time

overhead?

$$t = \max\left[t_i\right] + d = t_m + d \quad >> d$$

(min) cycle time

gate delay in stage i

delay in latches between stages
(= clock pulse, or clock cycle time)

max gate delay in stage

- Cycle time is the same for all stages
  - – time (in clock pulses) to execute the cycle
- Each stage executed in one cycle time
- Longest stage determines cycle time

Pipeline Speedup

n instructions, k stages

n instructions, k stages
τ = stage delay = cycle time

Time not pipelined: $T_1 = nkt$

(pessimistic because of assuming, that each stage would still have τ cycle time)

Time pipelined: $T_k = [k + (n-1)]t$

k cycles until 1st instruction completes

1 cycle for each of the rest (n-1) instructions

Pipeline Speedup (2)

n instructions, k stages

n instructions, k stages
τ = stage delay = cycle time

Time not pipelined: $T_1 = nk\tau$

(pessimistic because of assuming, that each stage would still have τ cycle time)

Time pipelined: $T_k = [k + (n-1)]\tau$

Speedup with k stages: $S_k = \frac{T_1}{T_k} = \frac{nk\tau}{[k+(n-1)]\tau} = \frac{nk}{[k+(n-1)]}$

Fig. 11.14

Branch Problem Solutions

- Delayed Branch
  - compiler places some useful instructions (1 or more!) after branch (or jump) instructions
  - these instructions are almost completely executed when branch decision is known
  - less actual work lost
  - can be difficult to do
  - conditional branches tricky, must be able to stop changes (by instruction in delay slot) in case there is no branch

Fig. 12.7

Branch Problem Solutions (contd)

- Multiple instruction streams
  - execute speculatively in both directions
    - Problem: we do not know the branch target address early!
  - if one direction splits, continue each way
  - lots of hardware
    - speculative results, control
  - speculative instructions may delay real work
    - bus & register contention?
  - need to be able to <u>cancel</u> not-taken instruction streams in pipeline

Branch Problem Solutions (contd)

- Prefetch Branch Target

IBM 360/91 (1967)

  - prefetch just branch target instruction
  - do not execute it, I.e., do only FI stage
  - if branch take, no need to wait for memory
- Loop Buffer
  - keep n most recently fetched instructions in high speed buffer inside CPU
  - works for small loops (at most n instructions)

Branch Problem Solutions (contd)

- Branch Prediction
  - guess (intelligently) which way branch will go
  - fixed prediction: take it, do not take it
  - based on opcode
    - E.g., BLE instruction *usually* at the end of loop?
  - taken/not taken prediction
    - based on previous time this instruction was executed
    - need space (1 bit) in CPU for each (?) branch
    - end of loop always wrong twice!
    - Extension based on two previous times
      - need more space (2 bits)

Fig. 11.16

Branch Address Prediction

- It is not enough to know whether branch is taken or not
- Must know also branch address to fetch target instruction
- Branch History Table
  - state information to guess whether branch will be taken or not
  - previous branch target address
  - stored in CPU for each (?) branch

Branch History Table

PowerPC 620

- Cached
  - entries only for most recent branches
    - Branch instruction address, or tag bits for it
    - Branch taken prediction bits (2?)
    - Target address (from previous time) or complete target instruction?
- Why cached
  - expensive hardware, not enough space for all possible branches
  - at lookup time check first whether entry for correct branch instruction

CPU Example: PowerPC

- User Visible Registers
  - 32 general purpose regs, each 64 bits
    - Exception reg (XER), 32 bits
  - 32 FP regs, each 64 bits
    - FP status & control (FPSCR), 32 bits
  - branch processing unit registers
    - Condition, 32 bits
      - 8 fields, each 4 bits
      - identity given in instructions
    - Link reg, 64 bits
      - E.g., return address
    - Count regs, 64 bits
      - E.g., loop counter

Fig. 11.22

Fig. 11.23a

Table 11.3

Fig. 11.23b

Table 11.4

CPU Example: PowerPC

- Interrupts
  - cause
    - system condition or event
    - instruction

Table 11.5

CPU Example: PowerPC

- Machine State Register, 64 bits Table 11.6
  - bit 48: external (I/O) interrupts enabled?
  - bit 49: privileged state or not
  - bits 52&55: which FP interrupts enabled?
  - bit 59: data address translation on/off
  - bit 63: big/little endian mode
- Save/Restore Regs SRR0 and SRR1
  - temporary data needed for interrupt handling

Power PC Interrupt Invocation

Table 11.6

- Save return PC to SRR0
  - current or next instruction at the time of interrupt
- Copy relevant areas of MSR to SRR1
- Copy additional interrupt info to SRR1
- Copy fixed new value into MSR
  - different for each interrupt
  - address translation off, disable interrupts
- Copy interrupt handler entry point to PC
  - two possible handlers, selection based on bit 57 of original MSR

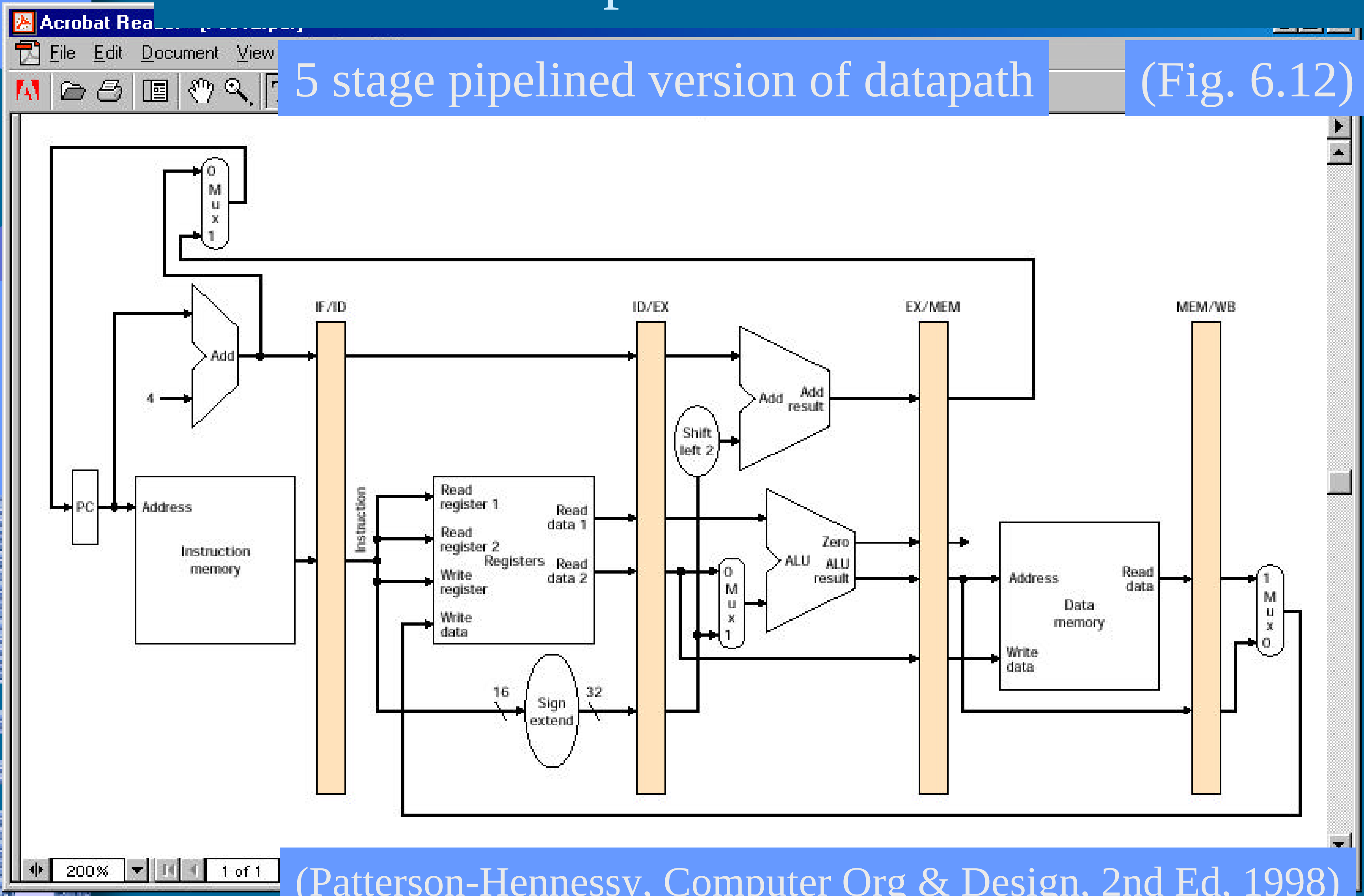
Power PC Interrupt Return

Table 11.6

- Return From Interrupt (rfi) instruction
  - privileged
- Rebuild original MSR from SRR1
- Copy return address from SRR0 to PC

-- End of Chapter 11: CPU Structure --

5 stage pipelined version of datapath (Fig. 6.12)



(Patterson-Hennessy, Computer Org & Design, 2nd Ed, 1998)