CPU Structure and Function
Ch 12

General Organisation

Registers

Instruction Cycle

Pipelining

Branch Prediction

Interrupts

General CPU Organization (4)

- ALU
  - does all real 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. 12.1 (Fig. 11.1 [Stal99])

Fig. 12.2 (Fig. 11.2 [Stal99])

More in Chapters 16-17 (Ch 14-15 [Stal99])

Register Organisation (4)

- 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 (6)

- Varies from one architecture to another
- General purpose registers (GPR)
  - Data, address, index, PC, condition, ….
- Data registers
  - Int, FP, Double, Index
- Address registers
- 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 (not 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. 12.7

(Fig. 11.7 [Stal99])

PSW - Program Status Word (6)

- State info from latest ALU-op
  - 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 (4)

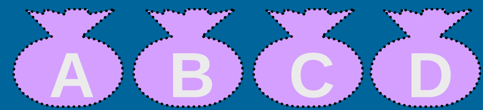
Fig. 11.4 [Stal99]

- Basic cycle with interrupt handling
- Indirect cycle — Figs 12.4-5 (Fig. 11.5-6 [Stal99])
- Data Flow — Figs 12.6-8 (Fig. 11.7-9 [Stal99])
  - CPU, Bus, Memory
- Data Path — Fig 16.5 (Fig. 14.5 [Stal99])
  - CPU's "internal data bus" or "data mesh" — Fig 3.1 [HePa96]
  - All computation is data transformations occurring on the data path
  - Control signals determine data flow & action for each clock cycle

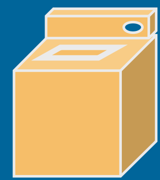
Pipeline Example

(liukuhihna)

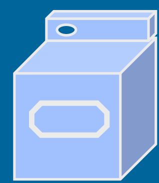
- Laundry Example (David A. Patterson)
- Ann, Brian, Cathy, Dave
  each have one load of clothes
  to wash, dry, and fold



- Washer takes 30 minutes

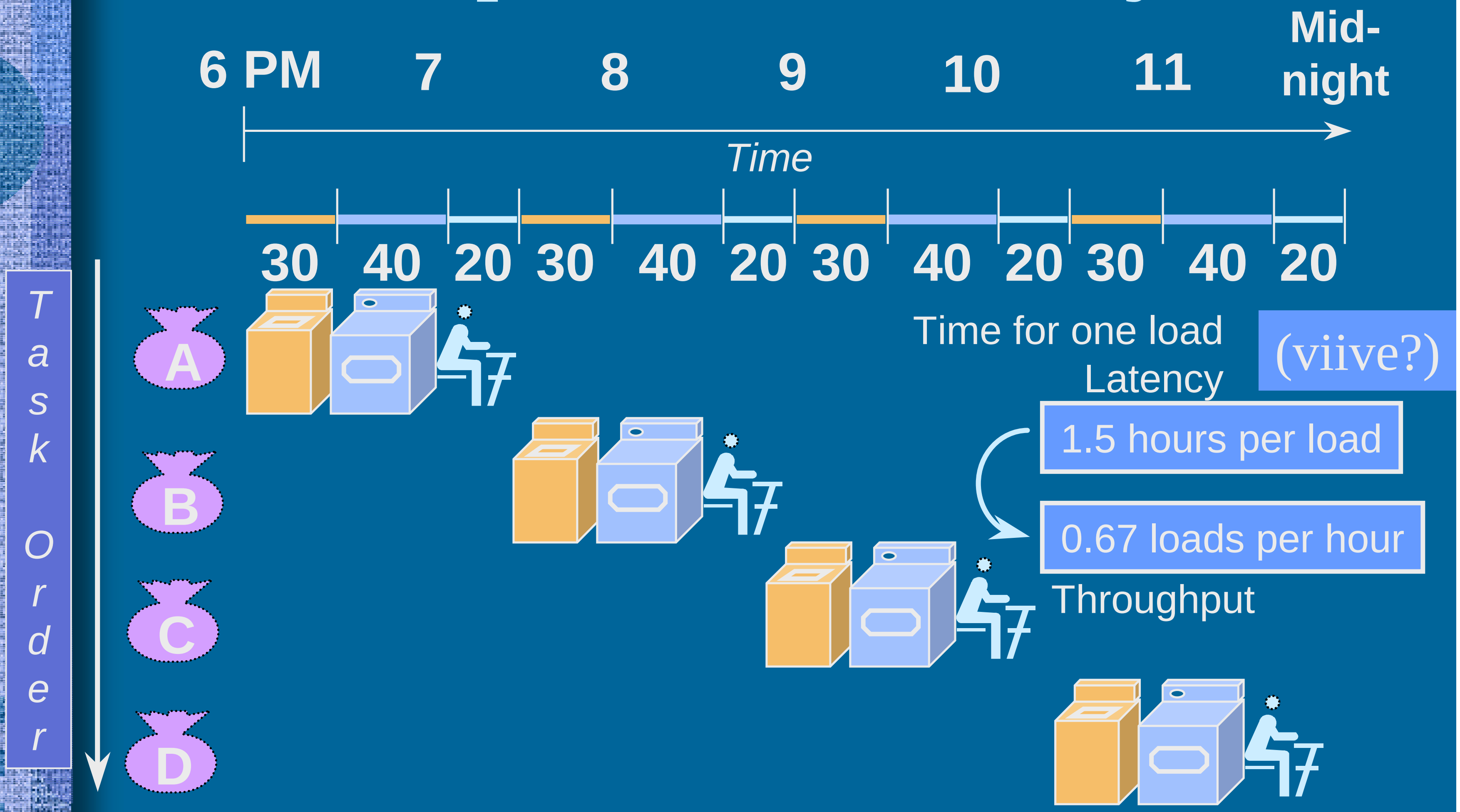


- Dryer takes 40 minutes



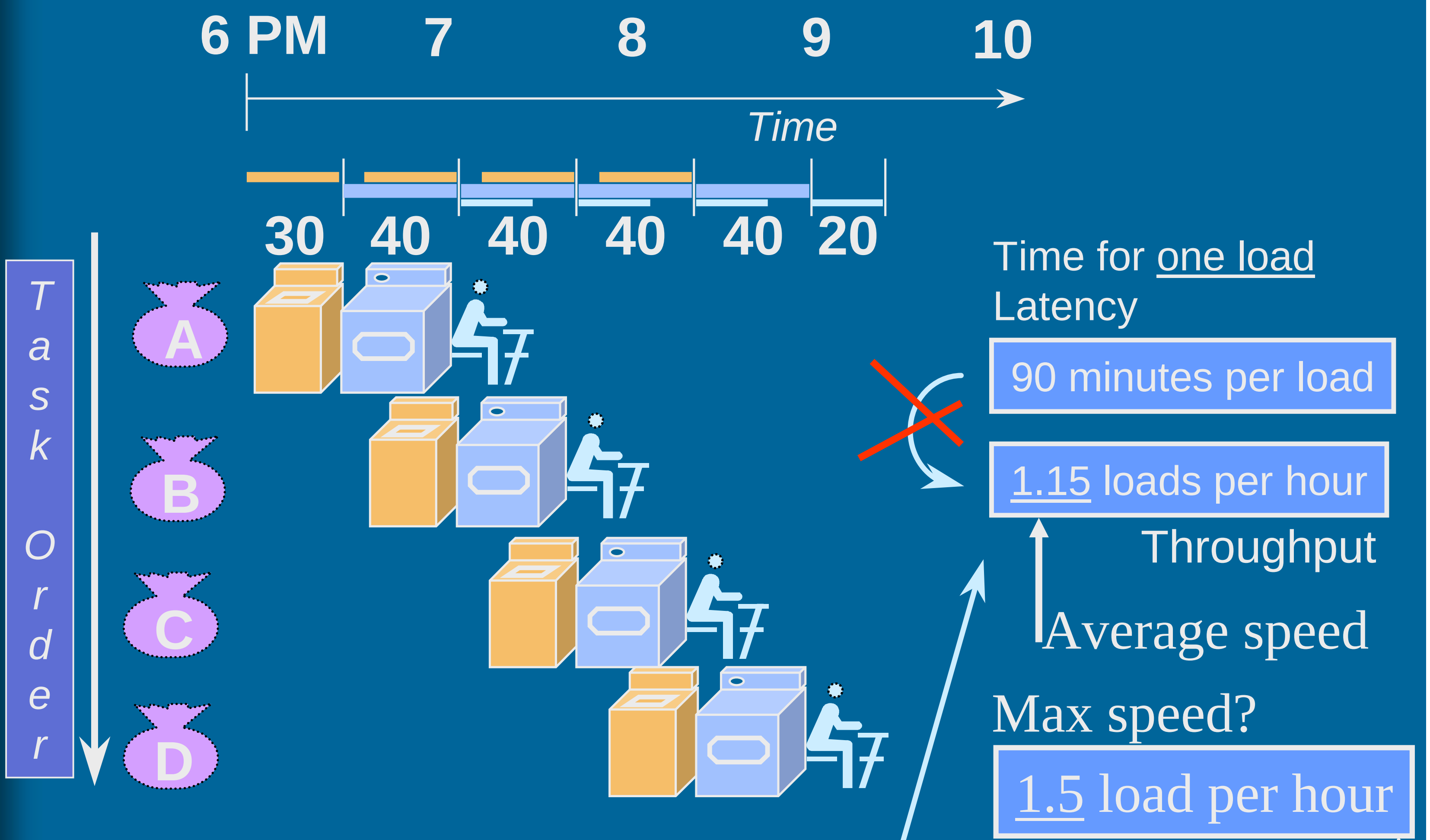
- “Folder” takes 20 minutes

Sequential Laundry (7)



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

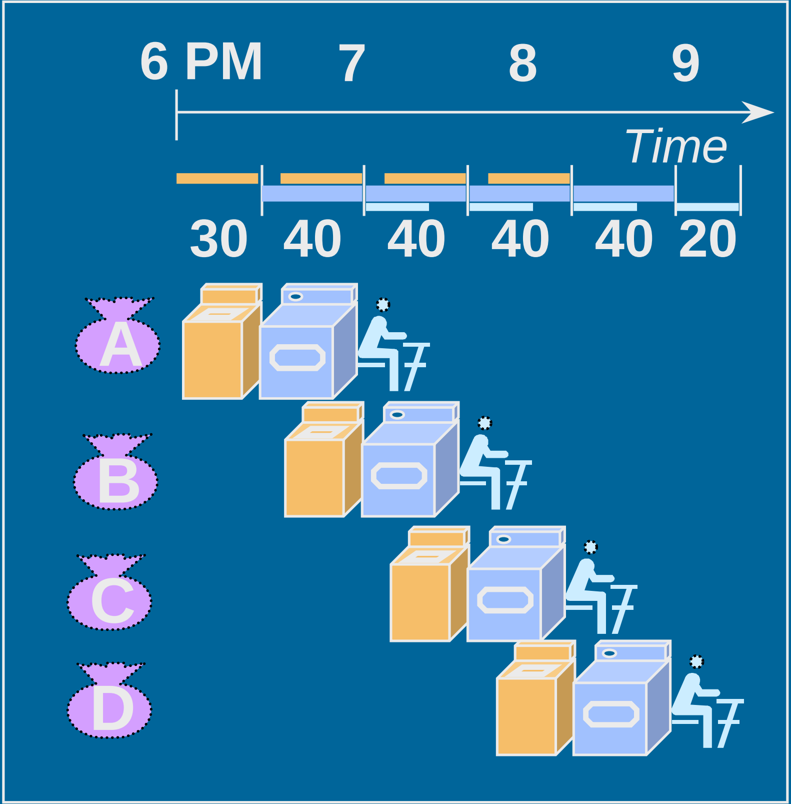
Pipelined Laundry (11)



- Pipelined laundry takes 3.5 hours for 4 loads
- At best case, laundry is completed every 40 minutes

Pipelining Lessons (4)

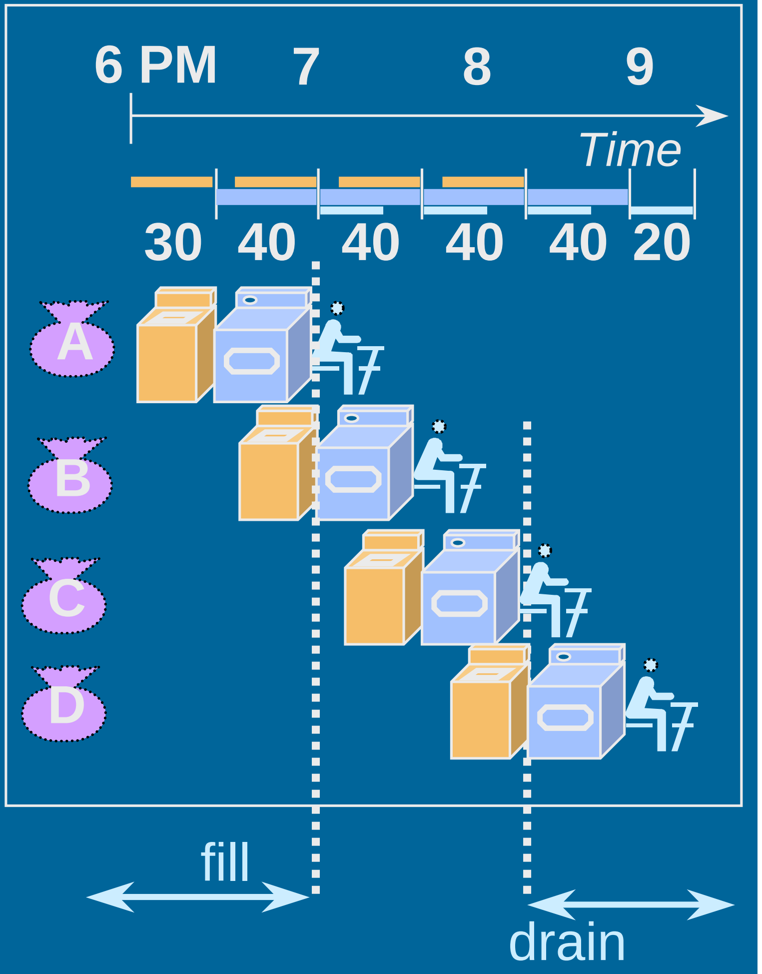
- Pipelining doesn't help latency of single task, but it helps throughput of the entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously
- Potential speedup
  = maximum possible speedup
  = Number pipe stages



(nopeutus)

Pipelining Lessons (3)

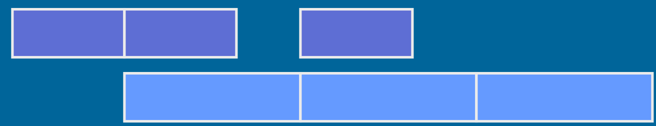
- Unbalanced lengths of pipe stages reduces speedup
- May need more resources
  - Enough electrical current to run both washer and dryer simultaneously?
  - Need to have at least 2 people present all the time?
- Time to “fill” pipeline and time to “drain” it reduces speedup



2-stage Instruction Execution Pipeline (4)

Fig. 12.9 (Fig. 11.10 [Stal99])

- Good: instruction pre-fetch at the same time as execution of previous instruction
- Bad: execution phase 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

Fig. 12.10 (Fig. 11.11 [Stal99])

Pipeline Speedup (6)

No pipeline, 9 instructions —9 * 6→ 54 time units

(Fig. 11.11 [Stal99])

6 stage pipeline, 9 instructions —Fig. 12.10→ 14 time units

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

(nopeutus)

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

Pipeline Execution Time (3)

- Time to execute one instruction , I.e., latency may be longer than for non-pipelined machine
  - extra latches to store intermediate results
- Time to execute 1000 instructions (seconds) is shorter (better) than that for non-pipelined machine, I.e., throughput (instructions per second) for pipelined machine is better (bigger) than that for non-pipelined machine
  - parallel actions speed-up overall work load
- 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. 12.11 (Fig. 11.12 [Stal99])

Fig. 12.12-13 (Fig. 11.13 [Stal99])

Pipeline Speedup Problems (3)

Fig. 12.11 (Fig. 11.12 [Stal99])

- Dependencies between instructions
  - data dependency
    - One instruction depends on data produced by some earlier instruction
  - structural dependency
    - Many instructions need the same resource at the same time
    - memory bus, ALU, …

value known after EI stage

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

value needed in CO stage

WO

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

FI

FO

memory bus use

Cycle Time (3)

overhead?

$$\tau = \max[\tau_i] + d = \tau_m + d \quad >> d$$

(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 stage
- Each stage takes one cycle time to execute
- Longest stage determines min cycle time
  - max MHz rate for system clock

Pipeline Speedup (1)

n instructions, k stages

Time
not pipelined: $T_1 = nk\tau$

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

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

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

k cycles until 1st instruction completes

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

Pipeline Speedup (1)

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:

$$T_k = \left[k + (n-1)\right]\tau$$

Speedup with k stages:

$$S_k = \frac{T_1}{T_k} = \frac{nk\tau}{\left[k + (n-1)\right]\tau} = \frac{nk}{\left[k + (n-1)\right]}$$

Fig. 12.14 (Fig. 11.14 [Stal99])

Branch Problem Solutions (5)

- 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
    - execute them always!
    - hopefully useful work
    - o/w NO-OP
  - less actual work lost
  - can be difficult to do

Fig. 13.7

(Fig. 12.7 [Stal99])

Branch Probl. Solutions (contd) (6)

- 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 again
  - lots of hardware
    - speculative results (registers!), control
  - speculative instructions may delay real work
    - bus & register contention?
    - Need multiple ALUs?
  - need to be able to <u>cancel</u> not-taken instruction streams in pipeline

Branch Probl. Solutions (contd) (2)

- 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 Probl. Solutions (contd) (4)

- Static Branch Prediction
  - guess (intelligently) which way branch will go
  - static prediction: all *taken* or all *not taken*
  - static prediction based on opcode
    - E.g., because BLE instruction is *usually* at the end of loop, guess "taken" for all BLE instructions

Branch Probl. Solutions (contd) (5)

- Dynamic branch prediction
  - based on previous time this instruction was executed
  - need a CPU "cache" of addresses of branch instructions, and taken/not taken information
    - 1 bit
  - end of loop always wrong twice!
  - extension: prediction based on two previous time executions of that branch instruction
    - need more space (2 bits)

Fig. 12.17

(Fig. 11.16 [Stal99])

Branch Address Prediction (3)

- 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 "cache" 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
    - Index/tag bits of branch instruction address

CPU Example: PowerPC

- User Visible Registers Fig. 12.23 (Fig. 11.22 [Stal99])
  - 32 general purpose regs, each 64 bits
    - Exception reg (XER), 32 bits Fig. 12.24a (Fig. 11.23a)
  - 32 FP regs, each 64 bits
    - FP status & control (FPSCR), 32 bits Table 12.3 (Tbl. 11.3)
  - branch processing unit registers
    - Condition, 32 bits Fig. 12.24b (Fig. 11.23b)
      - 8 fields, each 4 bits
      - identity given in instructions Table 12.4 (Tbl. 11.4)
    - Link reg, 64 bits
      - E.g., return address
    - Count regs, 64 bits
      - E.g., loop counter

CPU Example: PowerPC

- Interrupts
  - cause
    - system condition or event
    - instruction

Table 12.5

(Fig. 11.5 [Stal99])

CPU Example: PowerPC

(Tbl. 11.6 [Stal99])

Table 12.6

- Machine State Register, 64 bits
  - 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

(Tbl. 11.6 [Stal99])

Table 12.6

- 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

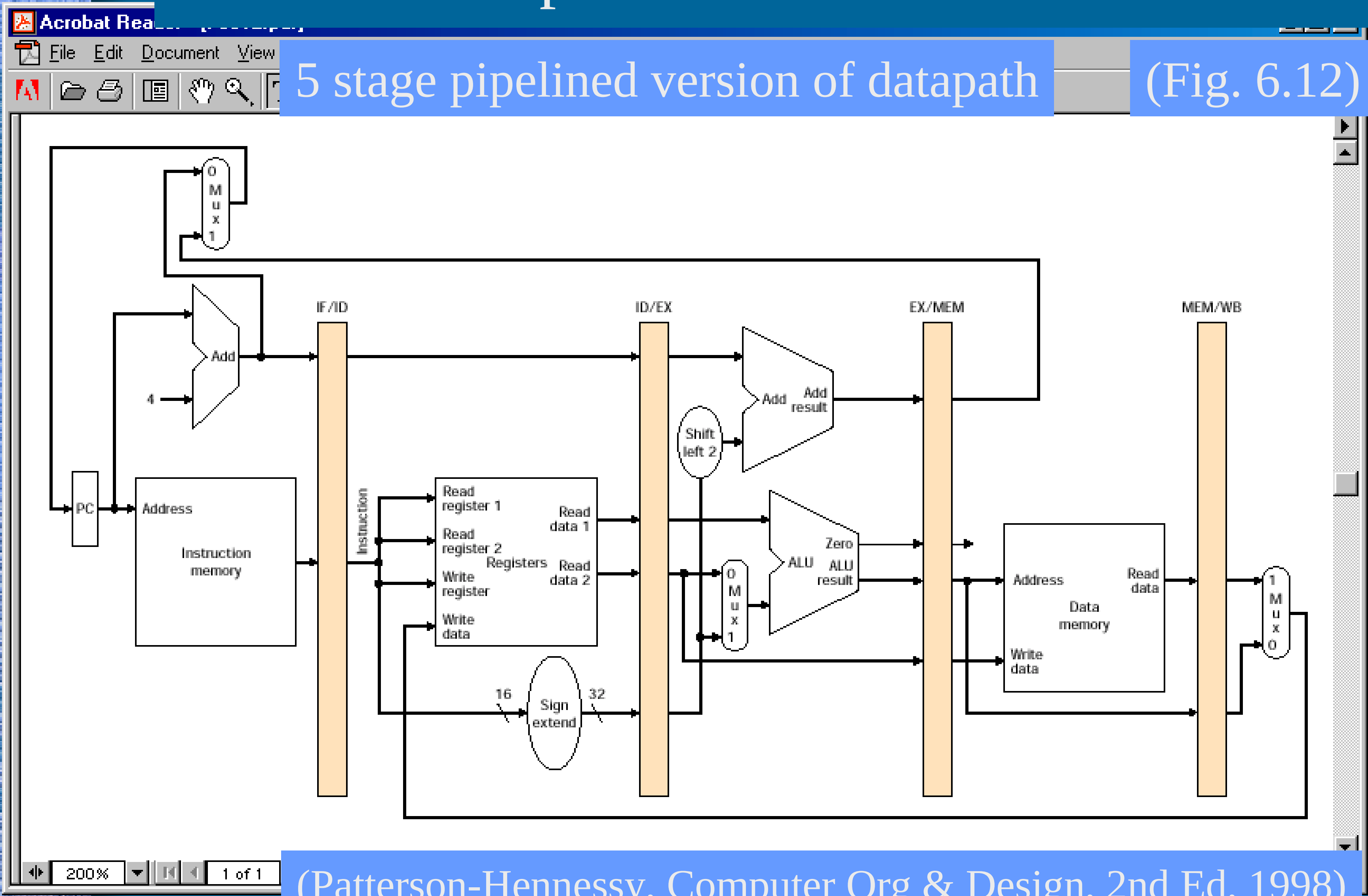
(Tbl. 11.6 [Stal99])

Table 12.6

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

-- End of Chapter 12: CPU Structure --

5 stage pipelined version of datapath (Fig. 6.12)



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