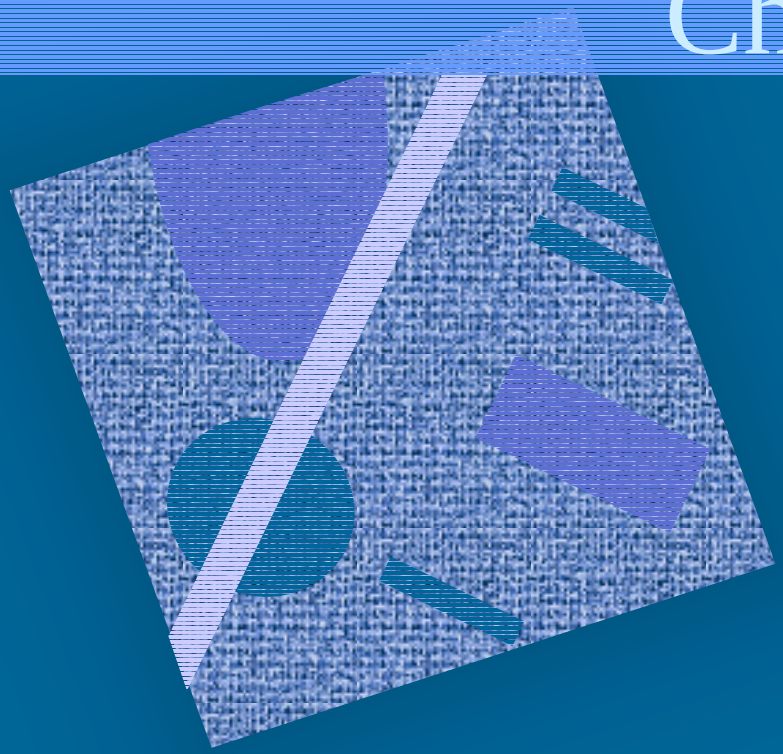
CPU Structure and Function
Ch 11



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. 11.1

Fig. 11.2

More in Chapters 14-15

Register Organisation (4)

- Registers make up CPU work space
- User visible registers

ADD R1,R2,R3

  - accessible directly via instructions
- Control and status registers

BNeq Loop

  - may be accessible indirectly via instructions
  - may be accessible only internally

HW exception

- Internal latches for temporary storage during instruction execution
  - E.g., ALU operand either from constant in instruction or from machine register

User Visible Registers

- Varies from one architecture to another
- General purpose register (GPR)
  - Data, address, index, PC, condition, ….
- Data register
  - Int, FP, Double, Index
- Address register
- 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. 11.7

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?

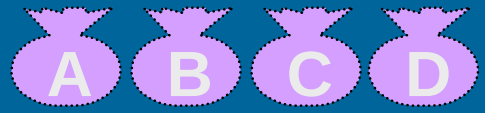
Instruction Cycle (4)

- Basic cycle with interrupt handling — Fig. 11.4
- Indirect cycle — Figs 11.5-6
- Data Flow — Figs 11.7-9
  - CPU, Bus, Memory
- Data Path — Fig 14.5
  - CPU's "internal data bus" or "data mesh"
  - All computation is data transformations occurring on the data path
  - Control signals determine data flow & action for each clock cycle

Pipeline Example

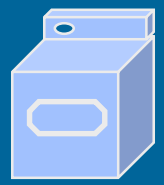
(liukuhihna)

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



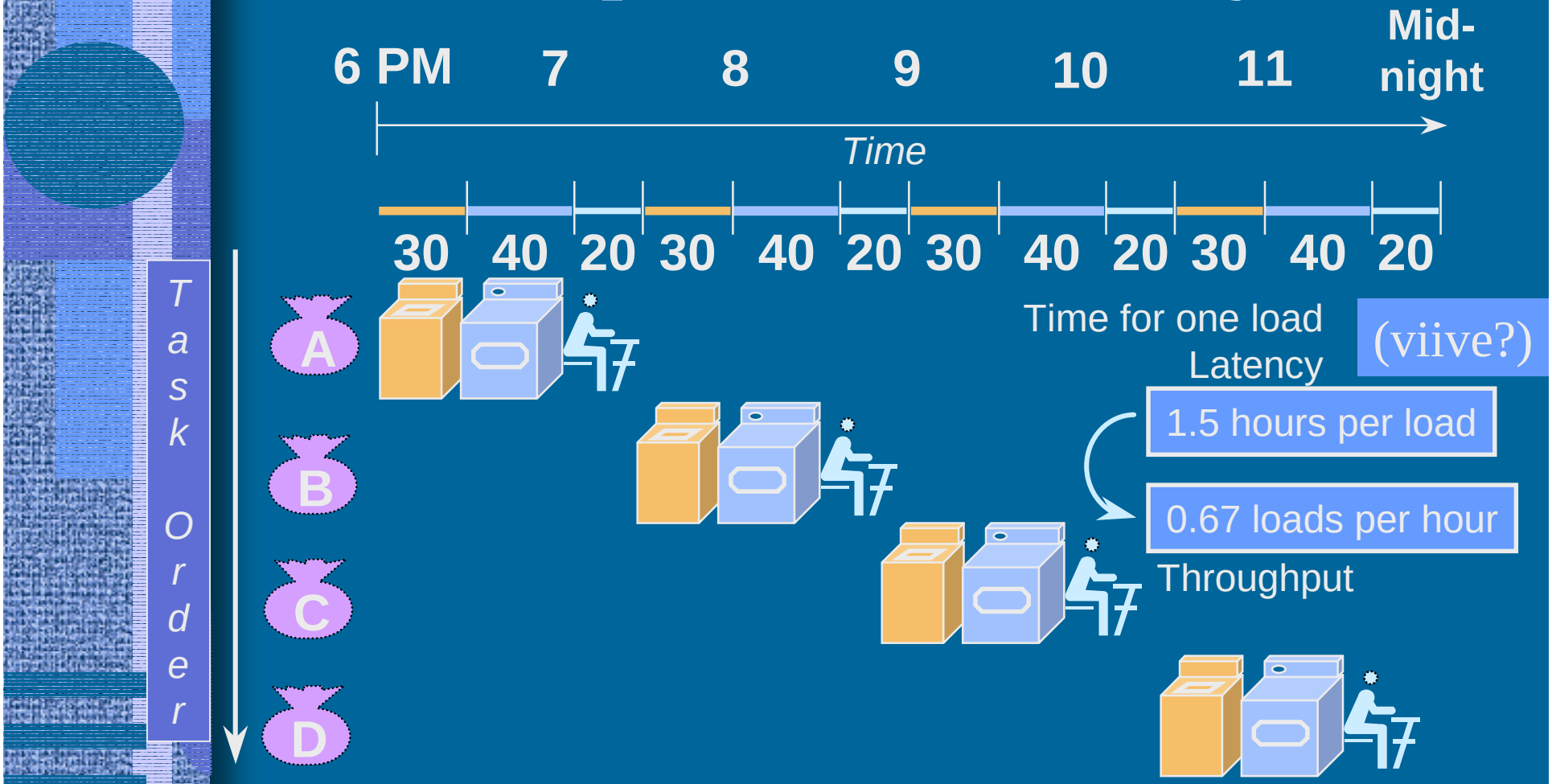
- Washer takes 30 minutes

- Dryer takes 40 minutes



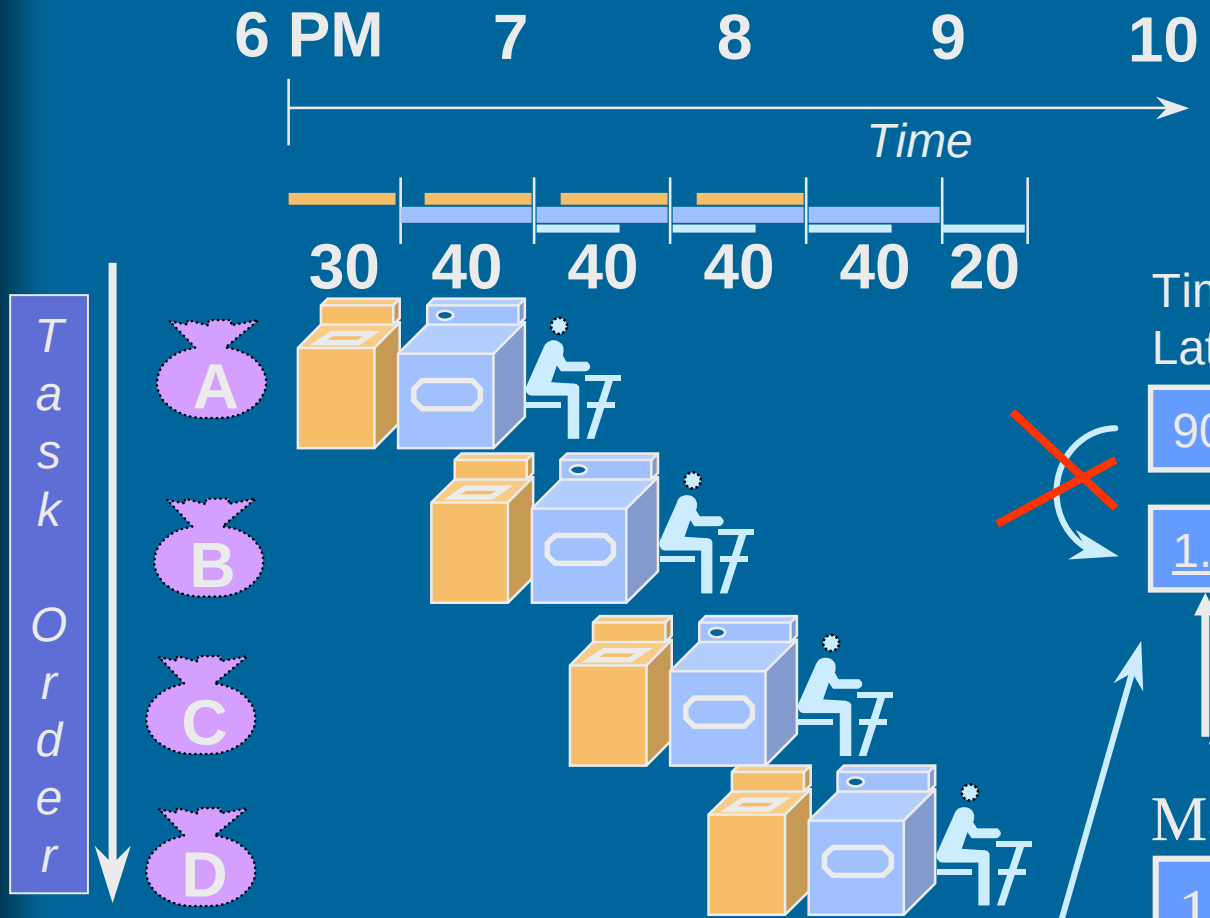
- “Folder” takes 20 minutes

Sequential Laundry (6)



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

Pipelined Laundry (11)



Time for one load
Latency

90 minutes per load

1.15 loads per hour

Throughput

Average speed

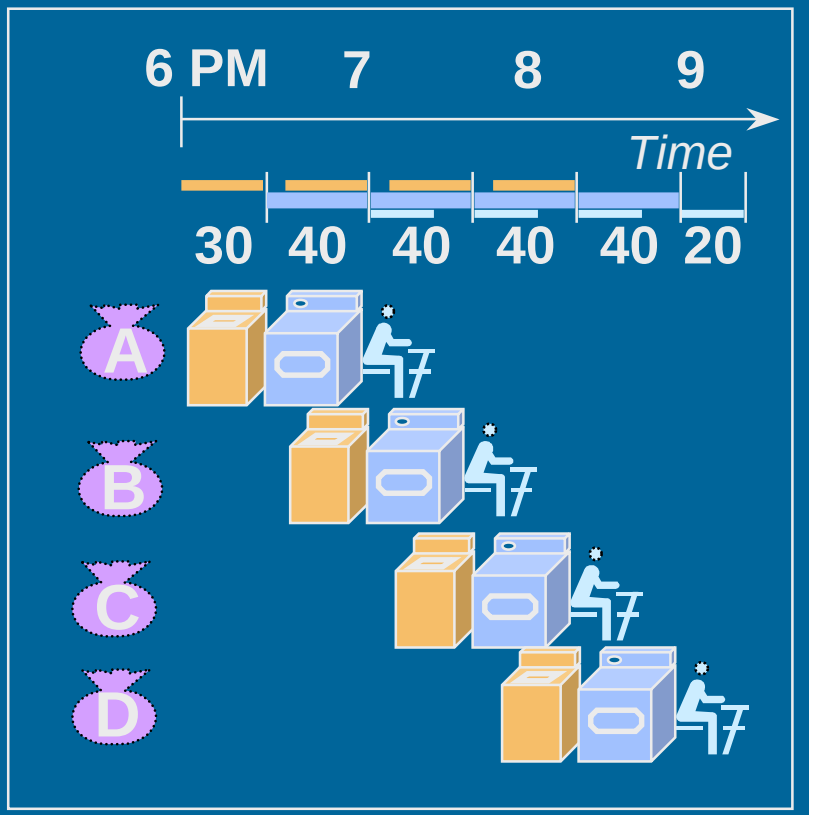
Max speed?

1.5 load per hour

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

Pipelining Lessons (4)

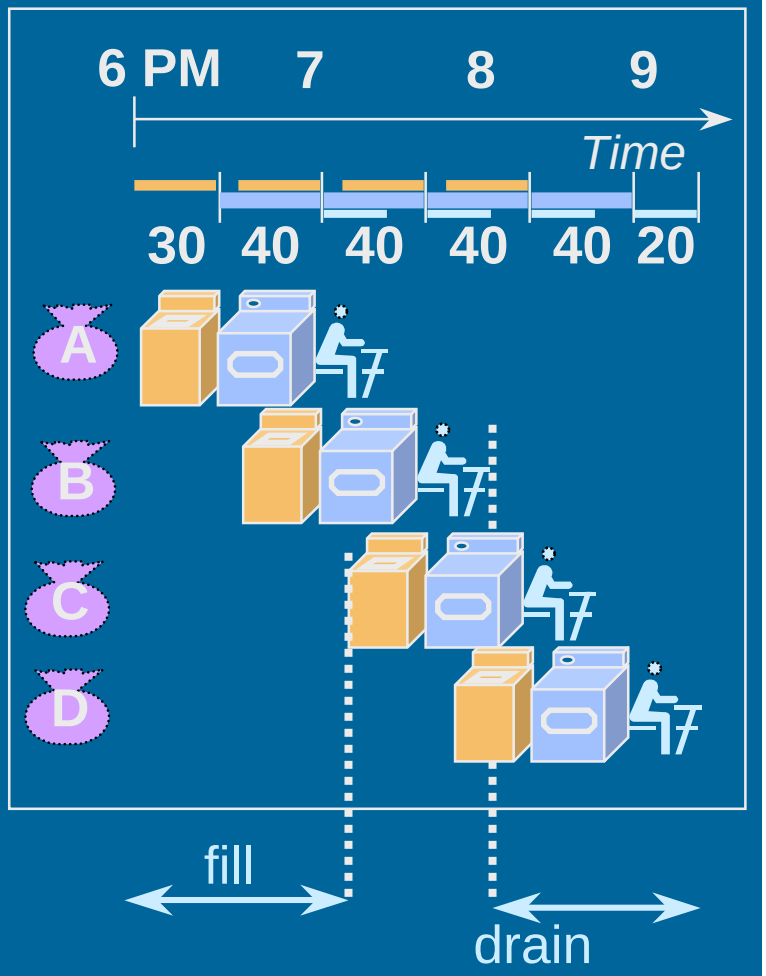
- Pipelining doesn't help latency of single task, but it helps throughput of the entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously
- Potential speedup
  = maximum possible speedup
  = Number pipe stages



(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. 11.10

- 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 6^{th} 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. 11.11

Pipeline Speedup (3)

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

6 stage pipeline, 9 instructions → Fig. 11.11 → 14 time units

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

(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 (latency, seconds) 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
- Is this good or bad? Why?

Pipeline Speedup Problems

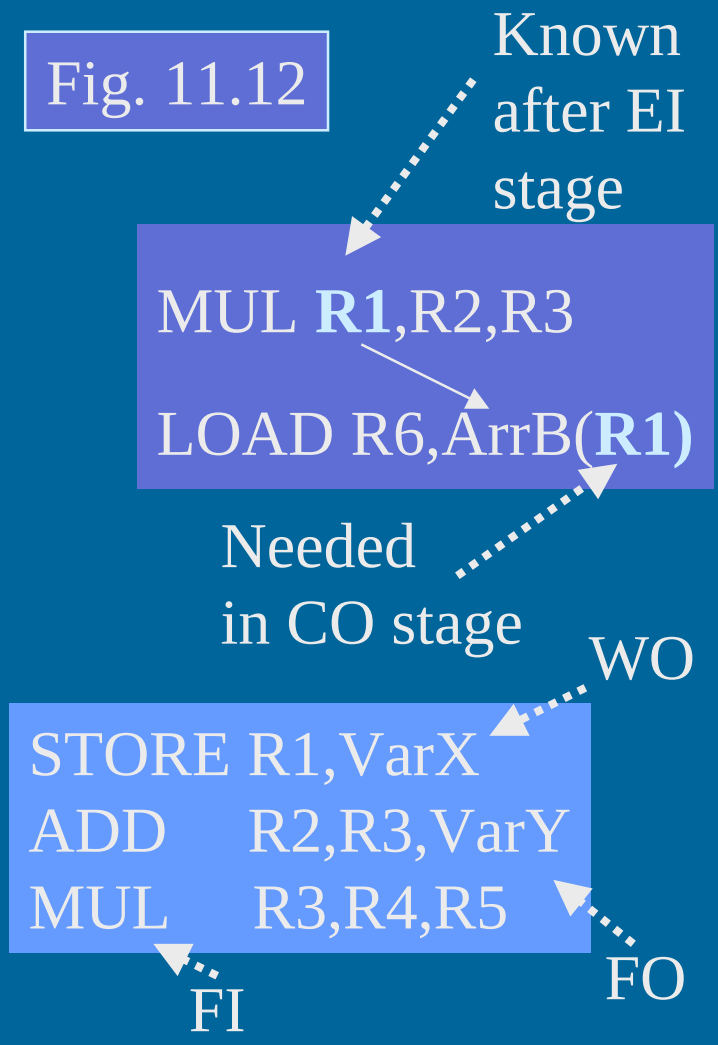
- Some stages are shorter than the others
- Dependencies between instructions
  - control dependency
    - E.g., conditional branch decision know only after EI stage

Fig. 11.12

Fig. 11.13

Pipeline Speedup Problems

- Dependencies between instructions
  - data dependency
    - 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, …



Cycle Time

overhead?

$$\tau = \max[\tau_i] + d = \tau_m + d \gg d$$

(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 min cycle time
  - max MHz rate for system clock

Pipeline Speedup

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

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

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 (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
  - less actual work lost
  - can be difficult to do

Fig. 12.7

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 to be able to cancel 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) (5)

- 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”
  - dynamic prediction taken/not taken
    - 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 time execution
      - need more space (2 bits)

Fig. 11.16

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 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

Fig. 11.22

  - 32 general purpose regs, each 64 bits
    - Exception reg (XER), 32 bits

Fig. 11.23a

  - 32 FP regs, each 64 bits
    - FP status & control (FPSCR), 32 bits

Table 11.3

  - branch processing unit registers
    - Condition, 32 bits
      - 8 fields, each 4 bits
      - identity given in instructions

Fig. 11.23b

Table 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 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

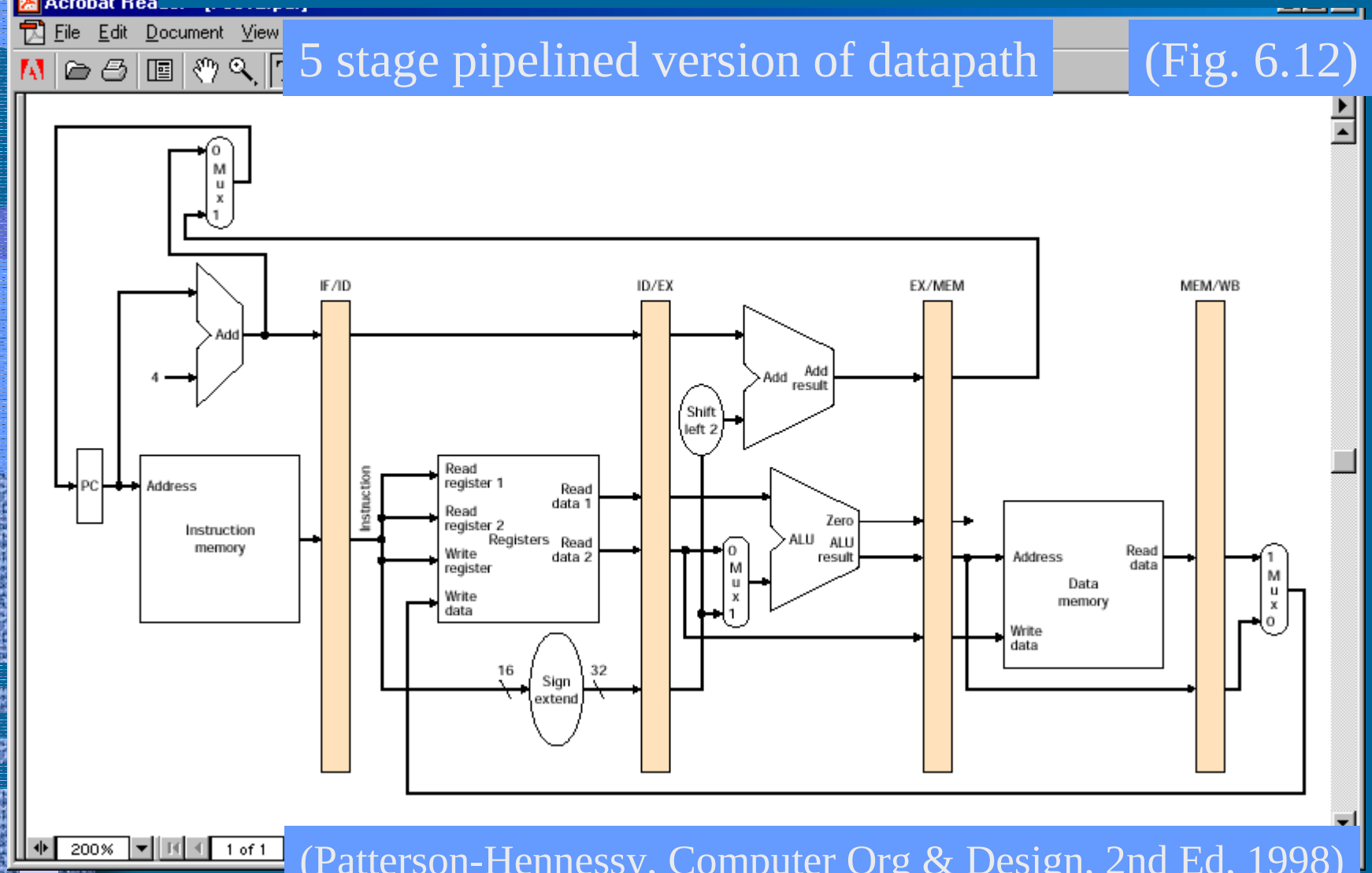
Power PC Interrupt Return

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)