

CPU Structure and Function

Ch 11

General Organisation
Registers
Instruction Cycle
Pipelining
Branch Prediction
Interrupts

04/10/2000

Copyright Teemu Kerola 2000

1

General CPU Organization ⁽⁴⁾

- ALU Fig. 11.1
 - does all real work
- Registers Fig. 11.2
 - data stored here
- Internal CPU Bus
- Control More in Chapters 14-15
 - determines who does what when
 - driven by clock
 - uses control signals (wires) to control what every circuit is doing at any given clock cycle

04/10/2000

Copyright Teemu Kerola 2000

2

Register Organisation ⁽²⁾

- 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

04/10/2000

Copyright Teemu Kerola 2000

3

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

04/10/2000

Copyright Teemu Kerola 2000

4

Control and Status Registers (5)

- PC
 - next instruction (not current!)
 - part of process state
- IR, Instruction (Decoding) Register Fig. 11.7
 - 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

04/10/2000

Copyright Teemu Kerola 2000

5

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?

04/10/2000

Copyright Teemu Kerola 2000

6

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

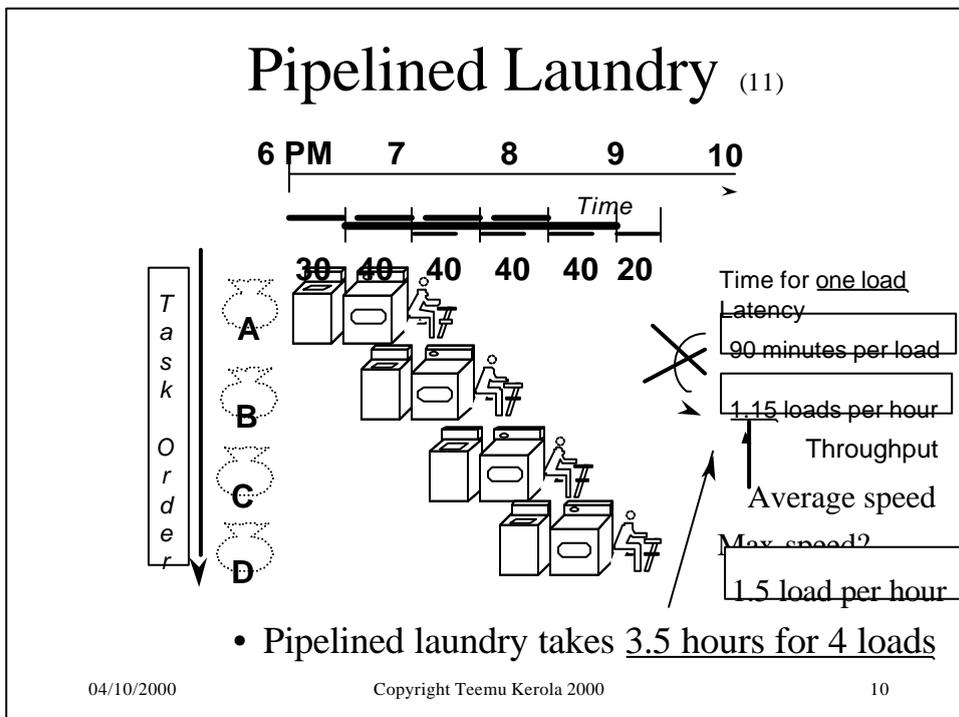
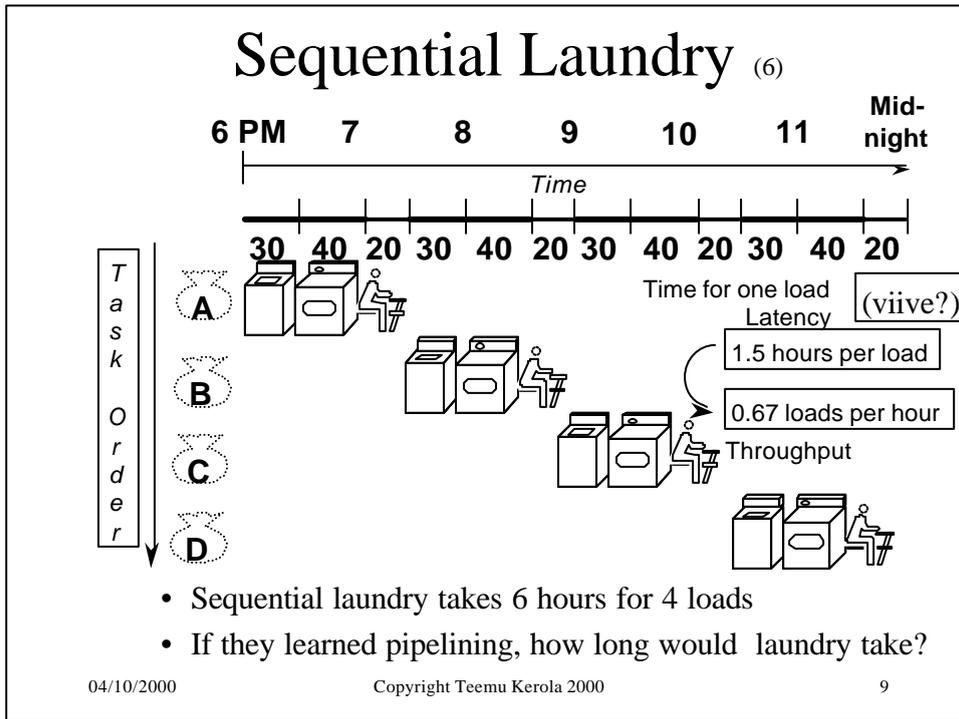
04/10/2000
Copyright Teemu Kerola 2000
7

Pipeline Example (liukuhinna)

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

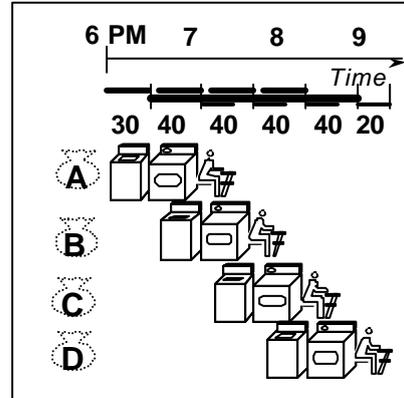
- Washer takes 30 minutes
- Dryer takes 40 minutes
- “Folder” takes 20 minutes

04/10/2000
Copyright Teemu Kerola 2000
8



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)

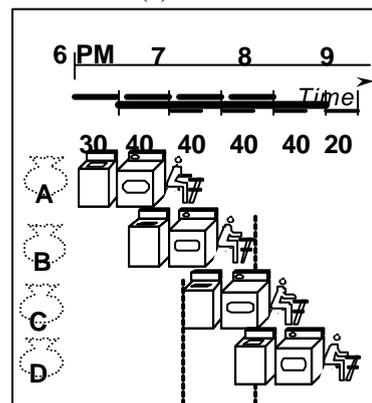
04/10/2000

Copyright Teemu Kerola 2000

11

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



fill drain

04/10/2000

Copyright Teemu Kerola 2000

12

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 6th instruction?
- Not enough parallelism \Rightarrow more stages?

04/10/2000

Copyright Teemu Kerola 2000

13

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

04/10/2000

Copyright Teemu Kerola 2000

14

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

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

04/10/2000
Copyright Teemu Kerola 2000
15

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?

04/10/2000
Copyright Teemu Kerola 2000
16

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

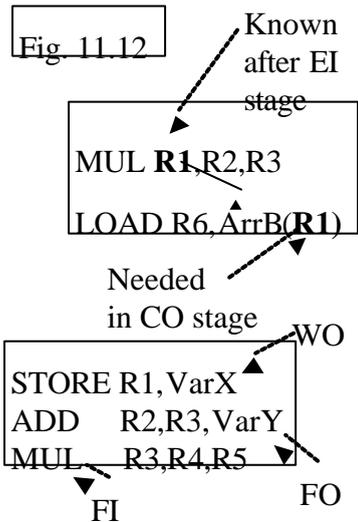
04/10/2000

Copyright Teemu Kerola 2000

17

Pipeline Speedup Problems

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



04/10/2000

Copyright Teemu Kerola 2000

18

Cycle Time

$$t = \max[t_i] + d = t_m + d \gg d$$

overhead?

↑ max gate delay in stage

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

↑ gate delay in stage i

(min) cycle time

- 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

04/10/2000
Copyright Teemu Kerola 2000
19

Pipeline Speedup

n instructions, k stages

n instructions, k stages
 $\tau = \text{stage delay} = \text{cycle time}$

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

04/10/2000
Copyright Teemu Kerola 2000
20

Pipeline Speedup ⁽¹⁾

n instructions, k stages

n instructions, k stages
 $\tau = \text{stage delay} = \text{cycle time}$

Time not pipelined: $T_1 = nkt$ (pessimistic because of assuming that each stage would still have τ cycle time)

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

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

Fig. 11.14

04/10/2000
Copyright Teemu Kerola 2000
21

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

04/10/2000
Copyright Teemu Kerola 2000
22

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
 - lots of hardware
 - speculative results, control
 - speculative instructions may delay real work
 - bus & register contention?
 - need to be able to cancel not-taken instruction streams in pipeline

04/10/2000

Copyright Teemu Kerola 2000

23

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)

04/10/2000

Copyright Teemu Kerola 2000

24

Branch Probl. Solutions (contd) ⁽⁵⁾

- 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 time execution
 - need more space (2 bits)

Fig. 11.16

04/10/2000

Copyright Teemu Kerola 2000

25

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

04/10/2000

Copyright Teemu Kerola 2000

26

Branch History Table

- **Cached** PowerPC 620
 - 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

04/10/2000

Copyright Teemu Kerola 2000

27

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 Fig. 11.23b
 - 8 fields, each 4 bits
 - identity given in instructions Table 11.4
 - Link reg, 64 bits
 - E.g., return address
 - Count regs, 64 bits
 - E.g., loop counter

04/10/2000

Copyright Teemu Kerola 2000

28

CPU Example: PowerPC

- Interrupts
 - cause
 - system condition or event
 - instruction

Table 11.5

04/10/2000

Copyright Teemu Kerola 2000

29

CPU Example: PowerPC

- 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

Table 11.6

04/10/2000

Copyright Teemu Kerola 2000

30

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

04/10/2000

Copyright Teemu Kerola 2000

31

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

04/10/2000

Copyright Teemu Kerola 2000

32

