

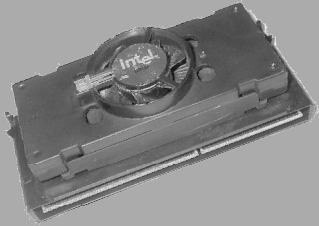
HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI

Lecture 7

CPU Structure and Function

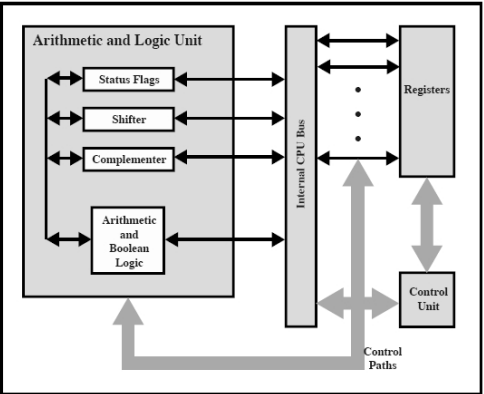
Ch 12 [Sta10]

- n Registers
- n Instruction cycle
- n Pipeline
- n Dependences
- n Dealing with Branches



General structure of CPU


- ALU
 - Calculations, comparisons
- Registers
 - Fast work area
- Processor bus
 - Moving bits
- Control Unit (*Ohjausyksikkö*)
 - What? Where? When?
 - Clock pulse
 - Generate control signals
 - What happens at the next pulse?
- MMU?
- Cache?



The diagram shows the internal structure of a CPU. It features an Arithmetic and Logic Unit (ALU) on the left, which contains Status Flags, a Shifter, a Complementer, and Arithmetic and Boolean Logic. These components are connected to a central Internal CPU Bus. To the right of the bus are Registers and a Control Unit. Bidirectional arrows indicate data flow between the ALU, Registers, and the bus. The Control Unit sends control signals to the bus. Below the bus, a Control Paths block is shown. At the bottom, the CPU connects to a System Bus, which is divided into Control Bus, Data Bus, and Address Bus.

(Sta10 Fig 12.1-2)


Computer Organization II, Autumn 2010, Teemu Kerola
22.11.2010 2



Registers

- Top of memory hierarchy
- User visible registers ADD R1,R2,R3
 - Programmer / Compiler decides how to use these
 - How many? Names?
- Control and status registers BNEQ Loop
 - Some of these used indirectly by the program
 - PC, PSW, flags, ...
 - Some used only by CPU internally
 - MAR, MBR, ...
- Internal latches (*apurekisteri*) for temporal storage during instruction execution
 - Example: Instruction register (IR) instruction interpretation; operand first to latch and only then to ALU
 - ALU output before result moved to some register

Computer Organization II, Autumn 2010, Teemu Kerola 22.11.2010 3



User visible registers

- Different processor families ⇒
 - different number of registers
 - different naming conventions (*nimeämistavat*)
 - different purposes
- General-purpose registers (*yleisrekisterit*)
- Data registers (*datarekisterit*) – not for addresses!
- Address registers (*osoiterekisterit*)
 - Segment registers (*segmentirekisterit*)
 - Index registers (*indeksirekisterit*)
 - Stack pointer (*pino-osoitin*)
 - Frame pointer (*ympäristöosoitin*)
- Condition code registers (*tilarekisterit*) No condition code regs.
IA-64, MIPS

Computer Organization II, Autumn 2010, Teemu Kerola 22.11.2010 4

Example

(Sta10 Fig 12.3)

Data Registers

D0	
D1	
D2	
D3	
D4	
D5	
D6	
D7	

Address Registers

A0	
A1	
A2	
A3	
A4	
A5	
A6	
A7	

Program Status

Program Counter
Status Register

(a) MC68000

General Registers

AX	Accumulator
BX	Base
CX	Count
DX	Data

Pointer & Index

SP	Stack Pointer
BP	Base Pointer
SI	Source Index
DI	Dest Index

Segment

CS	Code
DS	Data
SS	Stack
ES	Extra

Program Status

Instr Ptr
Flags

(b) 8086

Number of registers:
8,16, or 32 ok in 1980
RISC: several hundreds

General Registers

EAX	AX
EBX	BX
ECX	CX
EDX	DX

ESP	SP
EBP	BP
ESI	SI
EDI	DI

Program Status

FLAGS Register
Instruction Pointer

(c) 80386 - Pentium 4

Computer Organization II, Autumn 2010, Teemu Kerola 22.11.2010 5

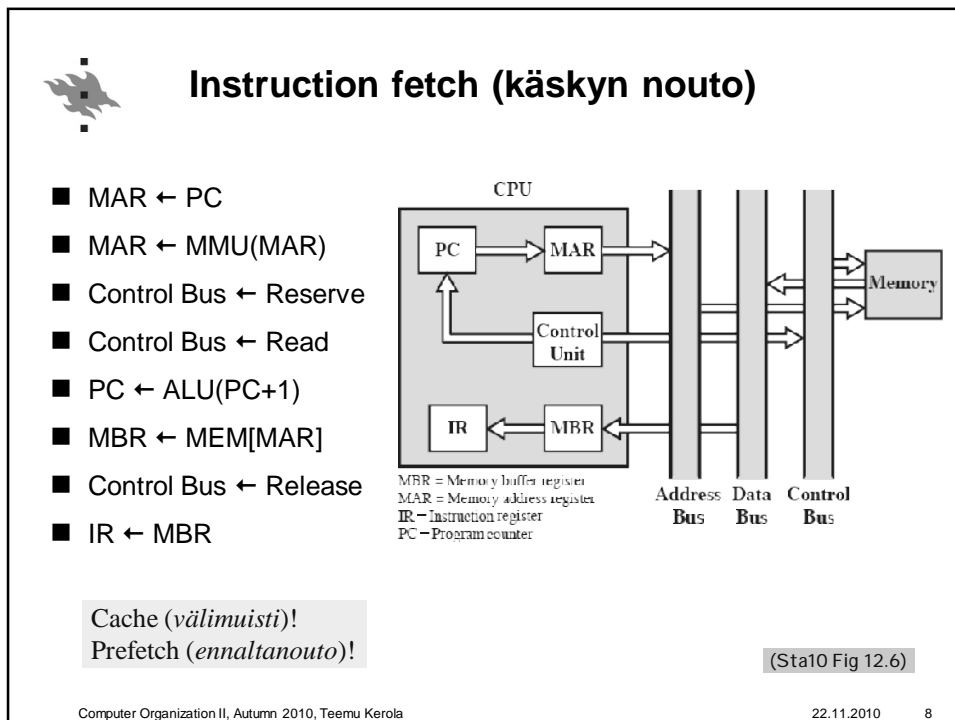
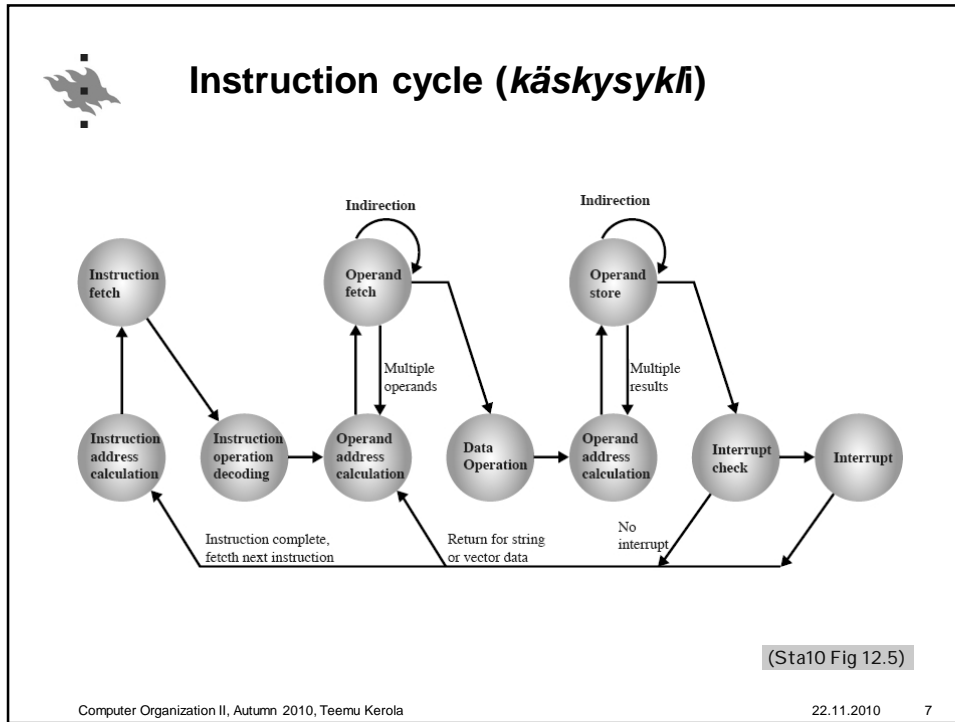
PSW - Program Status Word

- Name varies in different architectures
- State of the CPU
 - Privileged mode vs user mode
- Result of comparison (*vertailu*)
 - Greater, Equal, Less, Zero, ...
- Exceptions (*poikkeus*) during execution?
 - Divide-by-zero, overflow
 - Page fault, "memory violation"
- Interrupt enable/ disable
 - Each 'class' has its own bit
- Bit for interrupt request?
 - I/O device requesting guidance

Design issues:

- OS support
- memory and registers in control data storing
- paging
- subroutines and stacks
- etc

Computer Organization II, Autumn 2010, Teemu Kerola 22.11.2010 6



Operand fetch, Indirect addressing (Operandin nouto, epäsuora osoitus)

- MAR ← Address
- MAR ← MMU(MAR)
- Control Bus ← Reserve
- Control Bus ← Read
- MBR ← MEM[MAR]

- MAR ← MBR
- MAR ← MMU(MAR)
- Control Bus ← Read
- MBR ← MEM[MAR]
- Control Bus ← Release
- Cache!
- ALU? Regs? ← MBR

(Sta10 Fig 12.7)

Computer Organization II, Autumn 2010, Teemu Kerola 22.11.2010 9


Data flow, interrupt cycle

- MAR ← SP
- MAR ← MMU(MAR)
- Control Bus ← Reserve
- MBR ← PC
- Control Bus ← Write
- MAR ← SP ← ALU(SP+1)
- MAR ← MMU(MAR)
- MBR ← PSW
- Control Bus ← Write
- SP ← ALU(SP+1)
- PSW ← privileged & disable
- MAR ← Interrupt number
- Control Bus ← Read
- PC ← MBR ← MEM[MAR]
- Control Bus ← Release

← No address translation!

SP = Stack Pointer (Sta10 Fig 12.8)


Computer Organization II, Autumn 2010, Teemu Kerola 22.11.2010 10



Computer Organization II

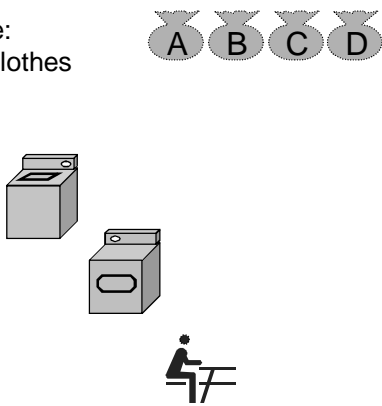
Instruction pipelining (*liukuhihna*)

Computer Organization II, Autumn 2010, Teemu Kerola 22.11.2010 11

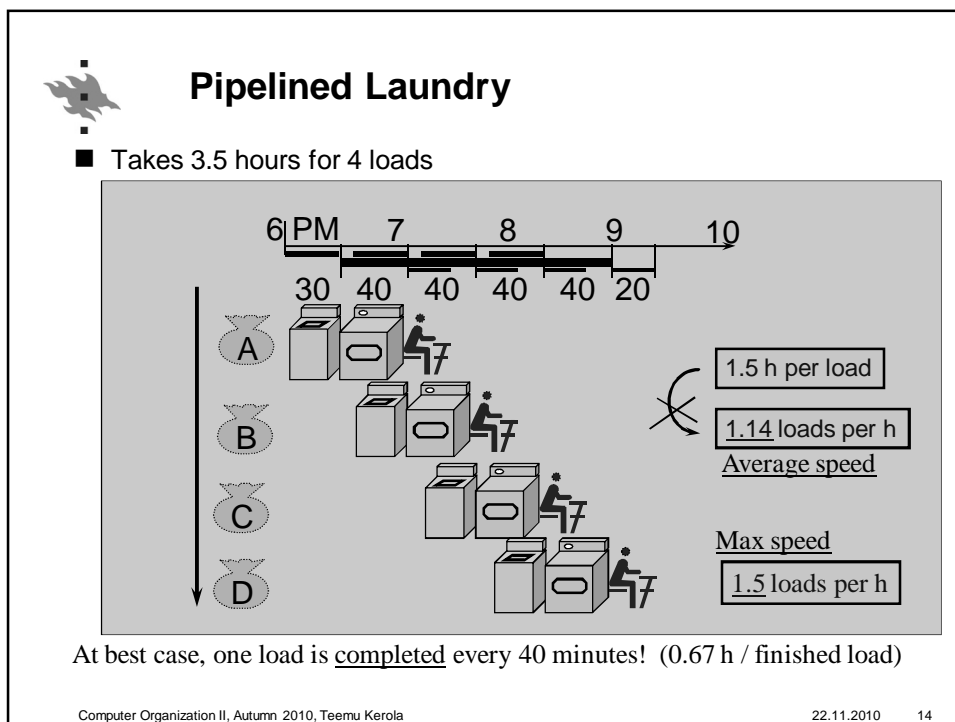
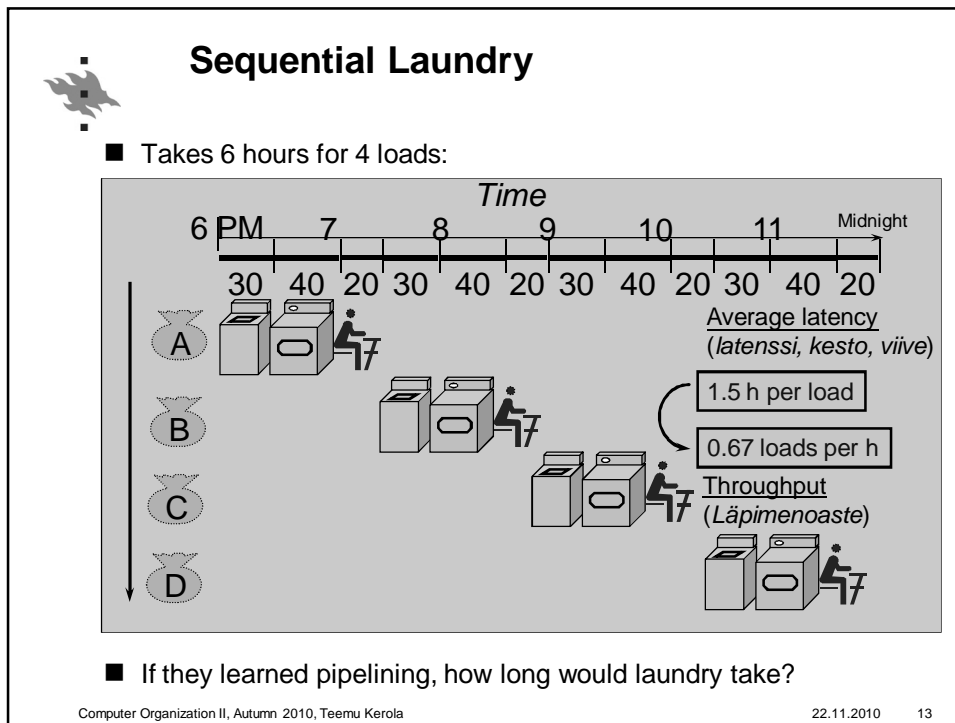


Laundry example (by David A. Patterson)

- Ann, Brian, Cathy, Dave:
each have one load of clothes
to wash, dry and fold
- Washer takes 30 min
- Dryer takes 40 min
- "Folder" takes 20 min



Computer Organization II, Autumn 2010, Teemu Kerola 22.11.2010 12



Lessons

- Pipelining does not help latency of single task, but it helps throughput of the entire workload
- Pipelining can delay single task compared with situation where it is alone in the system
 - Next stage occupied, must wait
- Multiple tasks operating simultaneously, but different phases
- Pipeline rate limited by slowest pipeline stage
 - Can proceed when all stages done
 - Not very efficient, if different stages have different durations, unbalanced lengths
- Potential speedup
 - = maximum possible speedup
 - = number of pipe stages

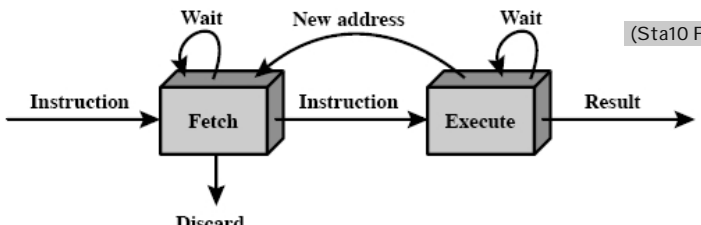
Computer Organization II, Autumn 2010, Teemu Kerola 22.11.2010 15

Lessons

- Complex implementation,
- May need more resources
 - Enough electrical current and sockets to use both washer and dryer simultaneously
 - Two (or three) people present all the time in the laundry
- Time to “fill” pipeline and time to “drain” it reduce speedup
 - Resources are not fully utilized
- “Hiccups” (*hikka*)
 - Variation in task arrivals, works best with constant flow of tasks

Computer Organization II, Autumn 2010, Teemu Kerola 22.11.2010 16

2-stage instruction execution pipeline (2-vaiheinen liukuhihna)



- Instruction prefetch (*ennaltanouto*) at the same time as execution of previous instruction
- Principle of locality (*paikallisuus*): assume 'sequential' execution
- Problems
 - Execution phase longer → fetch stage sometimes idle
 - Execution modifies PC (jump, branch) → fetched wrong instr.
 - Prediction of the next instruction's location was incorrect!
- Not enough parallelism → more stages?

Discussion?

Computer Organization II, Autumn 2010, Teemu Kerola 22.11.2010 17

6-Stage (6-Phase) Pipeline

	Time →													
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruction 1	FI	DI	CO	FO	EI	WO								
Instruction 2		FI	DI	CO	FO	EI	WO							
Instruction 3			FI	DI	CO	FO	EI	WO						
Instruction 4				FI	DI	CO	FO	EI	WO					
Instruction 5					FI	DI	CO	FO	EI	WO				
Instruction 6						FI	DI	CO	FO	EI	WO			
Instruction 7							FI	DI	CO	FO	EI	WO		
Instruction 8								FI	DI	CO	FO	EI	WO	
Instruction 9									FI	DI	CO	FO	EI	WO

FE - Fetch instruction

DI - Decode instruction

CO - Calculate operand addresses


FO - Fetch operands

EI - Execute instruction

WO - Write operand

(Sta10 Fig 12.10)


Computer Organization II, Autumn 2010, Teemu Kerola 22.11.2010 18



Pipeline speedup (*nopeutus*)?

- Lets calculate (based on Fig 12.10):
 - 6- stage pipeline, 9 instr. → 14 time units total
 - Same without pipeline → $9 \cdot 6 = 54$ time units
 - Speedup = $\text{time}_{\text{orig}} / \text{time}_{\text{pipeline}} = 54/14 = 3.86 < 6!$
 - Maximum speed at times 6-14
 - one instruction per time unit finishes
 - 8 time units → 8 instruction completions
 - Maximum speedup = $\text{time}_{\text{orig}} / \text{time}_{\text{pipeline}} = 48/8 = 6$
- Not every instruction uses every stage
 - Will not affect the pipeline speed – some stages unused
 - Speedup may be small (some stages idle, waiting for slow)
 - Unused stage → CPU idle (execution “bubble”)
 - Serial execution could be faster (no wait for other stages)

Computer Organization II, Autumn 2010, Teemu Kerola 22.11.2010 19



Pipeline performance: one cycle time

$$\tau = \max_{i=1..k} [\tau_i] + d = \tau_m + d \gg d$$

Cycle time (*jakson kesto*) Stage i time Latch delay, move data from one stage to next ~ one clock pulse Max time (duration) of the slowest stage (*Hitaimman vaiheen (max) kesto*)

- Cycle time is the same for all stages
 - Time (in clock pulses) to execute the stage
- Each stage takes one cycle time to execute
- Slowest stage determines the pace (*tahti, etenemisvauhti*)
 - The longest duration becomes bottleneck

Computer Organization II, Autumn 2010, Teemu Kerola 22.11.2010 20

Pipeline Speedup

n instructions, k stages, τ =cycle time

No pipeline: $T_1 = nk\tau$ Pessimistic: assumes the same duration for all stages

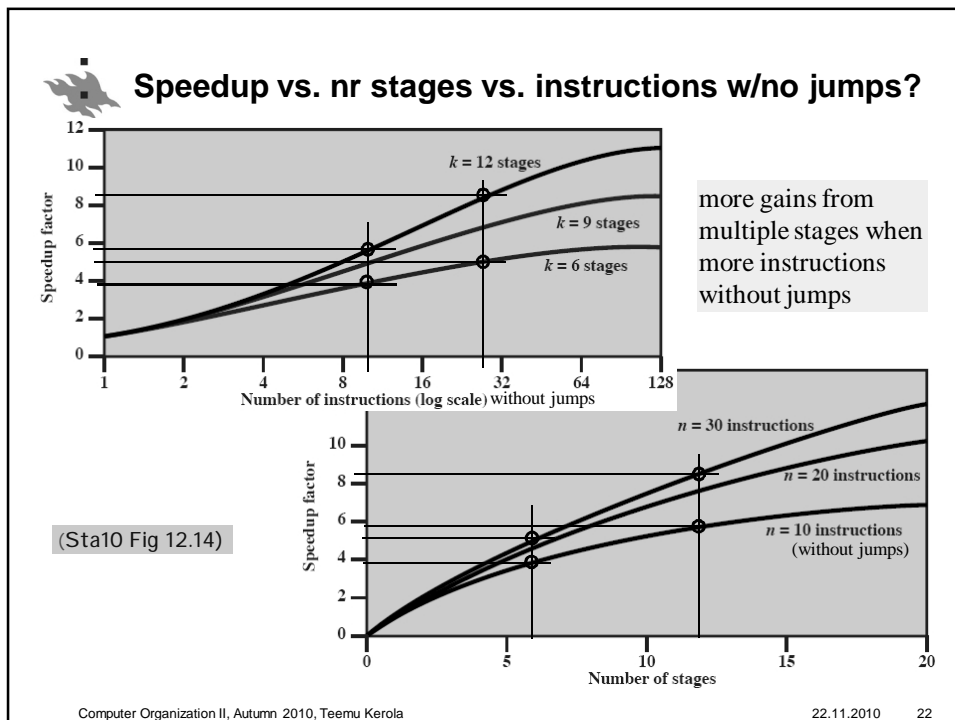
Pipeline: $T_k = [k + (n-1)]\tau$


\swarrow \searrow
 k stages before the first task (instruction) is finished next (n-1) tasks (instructions) will finish each during one cycle, one after another

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

See Sta10 Fig 12.10 and check yourself!

Computer Organization II, Autumn 2010, Teemu Kerola 22.11.2010 21






Pipeline Features

- Extra issues
 - CPU must store 'midresults' somewhere between stages and move data from buffer to buffer
 - From one instruction's viewpoint the pipeline takes longer time than single execution
- But still
 - Executing large set of instructions is faster
 - Better throughput (*läpimenoaste*) (instructions/sec)
- The parallel (concurrent) execution of instructions in the pipeline makes them proceed faster as whole, but slows down execution of single instruction

Computer Organization II, Autumn 2010, Teemu Kerola 22.11.2010 23



Pipeline Problems and Design Issues

- **Structural dependency** (*rakenteellinen riippuvuus*)
 - Several stages may need the same HW
 - Memory: FI, FO, WO
 - ALU: CO, EI

STORE	R1, VarX
ADD	R2, R3, VarY
MUL	R3, R4, R5
- **Control dependency** (*kontrolliriippuvuus*)
 - No knowledge on next instruction
 - E.g., (conditional) branch destination may be known only after EI-stage
 - → Prefetched wrong instructions

ADD	R1, R7, R9
Jump	There
ADD	R2, R3, R4
MUL	R1, R4, R5
- **Data dependency** (*datariippuvuus*)
 - Instruction needs the result of the previous non-finished instruction

MUL	R1, R2, R3
LOAD	R6, Arr(R1)

Computer Organization II, Autumn 2010, Teemu Kerola 22.11.2010 24



Pipeline Dependency Problem Solutions

- In advance: prevent (some) dependency problems completely
- At run time: Hardware must notice and wait until all dependencies are cleared
 - Add extra waits, “bubbles”, to the pipeline; Commonly used
 - Bubble (*kupla*) delays everything behind it in all stages
- Structural dependency
 - More hardware, e.g., separate ALUs for CO and EI stages
 - Lots of registers, less operands from memory
- Control dependency
 - Clear pipeline, fetch new instructions
 - Branch prediction, prefetch these or those?
- Data dependency
 - Change execution order of instructions
 - By-pass (*oikopolku*) in hardware between stages: earlier instruction's result can be accessed already before its WO-stage is done

Computer Organization II, Autumn 2010, Teemu Kerola

22.11.2010 25



Data dependency

- Read after Write (RAW) (a.k.a true or flow dependency)
 - Occurs if succeeding read takes place before the preceding write operation is complete
- Write after Read (WAR) (a.k.a antidependency)
 - Occurs if the succeeding write operation completes before the preceding read operation takes place
- Write after Write (WAW) (a.k.a output dependency)
 - Occurs when the two write operations take place in the reversed order of the intended sequence
- The WAR and WAW are possible only in architectures where the instructions can finish in different order

```
Load r1, A
Add r3, r2, r1
```

```
Add r3, r2, r1
Load r1, A
```

```
Add r1, r5, r6
Store r1, A
Add r1, r2, r3
```

Discussion?

Computer Organization II, Autumn 2010, Teemu Kerola

22.11.2010 26

Example: Data Dependency - RAW

Dependency: wait

```

        MUL R1, R2, R3
        ADD R4, R5, R6
        SUB R7, R1, R8
        ADD R1, R1, R3
    
```

Dependency: no wait

```

        MUL R1, R2, R3
        ADD R4, R5, R6
        SUB R7, R7, R8
        ADD R1, R1, R3
    
```

1	2	3	4	5	6	7	8	9	10	11
FI	DI	CO	FO	EI	WO					
	FI	DI	CO	FO	EI	WO				
		FI	DI	CO	WO	FO	EI	WO		
			FI	DI	CO	WO	FO	EI	WO	

1	2	3	4	5	6	7	8	9	10	11
FI	DI	CO	FO	EI	WO					
	FI	DI	CO	FO	EI	WO				
		FI	DI	CO	FO	EI	WO			
			FI	DI	CO	FO	EI	WO		

Discussion?

Computer Organization II, Autumn 2010, Teemu Kerola 22.11.2010 27

Example: Change instruction execution order

Need bubble

```

        MUL R1, R2, R3
        ADD R4, R5, R6
        SUB R7, R1, R8
        ADD R9, R0, R8
    
```

No effective dependencies

```

        MUL R1, R2, R3
        ADD R4, R5, R6
        ADD R9, R0, R8
        SUB R7, R1, R8
    
```

1	2	3	4	5	6	7	8	9	10	11
FI	DI	CO	FO	EI	WO					
	FI	DI	CO	FO	EI	WO				
		FI	DI	CO	WO	FO	EI	WO		
			FI	DI	CO	WO	FO	EI	WO	

1	2	3	4	5	6	7	8	9	10	11
FI	DI	CO	FO	EI	WO					
	FI	DI	CO	FO	EI	WO				
		FI	DI	CO	FO	EI	WO			
			FI	DI	CO	FO	EI	WO		

switched instructions

Computer Organization II, Autumn 2010, Teemu Kerola 22.11.2010 28

Example: By-pass (oikopolut)

- New wires (and temp registers, latches) in pipeline
 - E.g., instr. result available to FO phase directly from phase EI

MUL R1, R2, R3
 ADD R4, R5, R1
 SUB R7, R4, R1

1	2	3	4	5	6	7	8	9	10	11	
FI	DI	CO	FO	EI	WO				no by-pass		
	FI	DI	CO			FO	EI	WO			
		FI	DI	CO					FO	EI	WO

MUL R1, R2, R3
 ADD R4, R5, R1
 SUB R7, R4, R1

1	2	3	4	5	6	7	8	9	10	11
FI	DI	CO	FO	EI	WO					
	FI	DI	CO		FO	EI	WO		With by-pass	
		FI	DI	CO			FO	EI	WO	

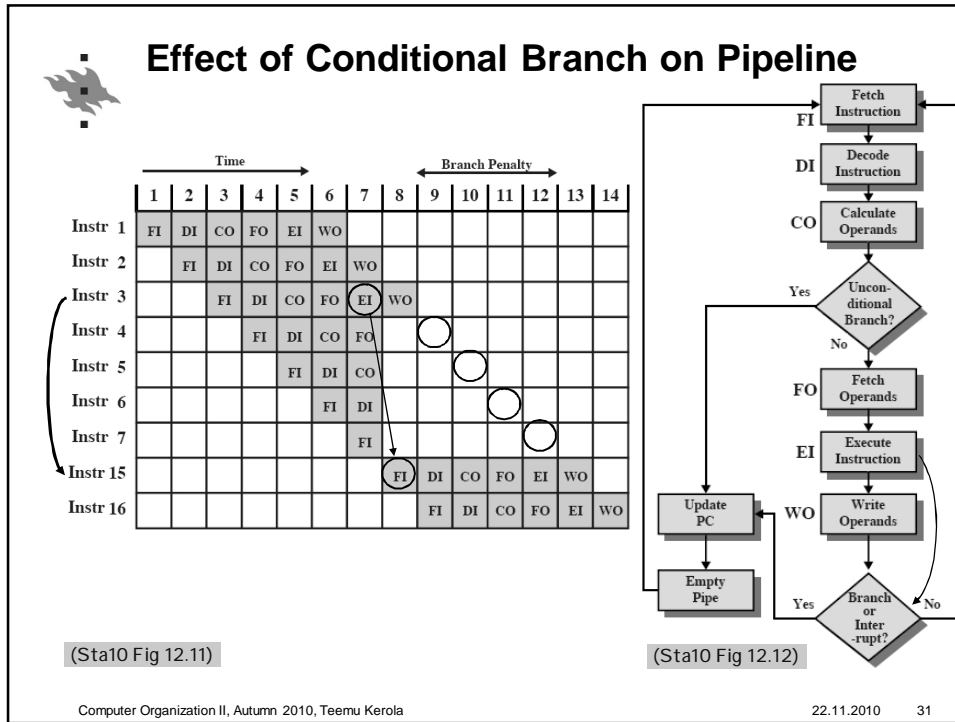
Computer Organization II, Autumn 2010, Teemu Kerola 22.11.2010 29

Computer Organization II

Pipelining and Jump Optimization

- Multiple streams (*Monta suorituspolkua*)
- Delayed branch (*Viivästetty hyppy*)
- Prefetch branch target (*Kohteen ennaltanouto*)
- Loop buffer (*Silmukkapuskuri*)
- Branch prediction (*Ennustuslogiikka*)

Computer Organization II, Autumn 2010, Teemu Kerola 22.11.2010 30



Delayed Branch (viivästetty haarautuminen)

```

sub r5, r3, r7
add r1, r2, r3
jump There
...
↓
sub r5, r3, r7
jump There
add r1, r2, r3
...
                
```

delay slot

- Compiler places some useful instructions (1 or more) after branch instructions (to delay slots)
 - Instruction in delay slots are always executed!
 - No roll-back of instructions needed due incorrect prediction
 - Rollback is difficult to do
 - If no useful instruction available, compiler uses NOP
- Less actual work lost if branch occurs
 - Next instruction almost done, when branch decision known
- This is easier than emptying the pipeline during branch
- Worst case: NOP-instructions wait some cycles
- Can be difficult to do (for the compiler)

Computer Organization II, Autumn 2010, Teemu Kerola 22.11.2010 32



Multiple instruction streams (*monta suorituspolkua*)

- Execute speculatively to both directions
 - Prefetch instructions that follow the branch to the pipeline
 - Prefetch instructions from branch target to (another) pipeline
 - After branch decision: reject the incorrect pipeline (its results)
- Problems
 - Branch target address known only after some calculations
 - Second split on one of the pipelines
 - Continue any way? Only one speculation at a time?
 - More hardware!
 - More pipelines, speculative results (registers!), control
 - Speculative instructions may delay real work
 - Bus and register contention? More ALUs?
- Capability to *cancel* not-taken instruction stream from pipeline
 - easier, if all changes done in WB phase

IBM 370/168,
IBM 3033
.....
Intel IA-64



Prefetch branch target (*kohteen ennaltanouto*)

- Prefetch just branch target instruction, but do not execute it yet
 - Do only FI-stage
 - If branch taken, no need to wait for memory
- Must be able to clear the pipeline
- Prefetching branch target may cause page-fault

IBM 360/91 (1967)



Loop buffer (*silmukkapuskuri*)

- Keep n most recently fetched instructions in high speed buffer inside the CPU
 - Use prefetch also
 - With good luck the branch target is in the buffer
 - F.ex. IF-THEN and IF-THEN-ELSE structures
- Works for small loops (at most n instructions)
 - Fetch from memory just once
- Gives better spacial locality than just cache

CRAY-1
 Motorola 68010
 ...
 Intel Core-2



Static Branch Prediction

- Make an (educated?) guess which direction is more probable:

Branch or no?

- Static prediction (*staattinen ennustus*)
 - Fixed: Always taken (*aina hypätään*)
 - Fixed: Never taken (*ei koskaan hypätä*)
 - ~ 50% correct
 - Predict by opcode (*operaatiokoodin perusteella*)
 - In advance decided which codes are more likely to branch
 - For example, BLE instruction is commonly used at the end of stepping loop, guess a branch
 - ~ 75% correct [LILJ88]

Motorola 68020
 VAX 11/780
 ...
 Intel Pentium III

Dynamic Branch Prediction

- **Dynamic prediction**
 - Make a guess based on earlier history for (this) branch
 - Logic: What has happened in the recent history with this instruction
 - Improves the accuracy of the prediction
 - Implementation: extra internal memory = branch history table
 - Instruction address (for this branch)
 - Branch target (instruction or address) – need this for quick action
 - Decision: taken / not taken
- **Simple prediction based on just the previous execution**
 - 1 bit memory is enough
 - Loops will always have one or two incorrect predictions

Cache?

Computer Organization II, Autumn 2010, Teemu Kerola
22.11.2010 37

2-Bit Branch Prediction Logic for One Instruction

- **Improved simple model**
 - Don't change the prediction with one misprediction
 - Based on two previous executions of this instruction
 - 2 bits enough

PowerPC 620
(Sta10 Fig 12.19)

Computer Organization II, Autumn 2010, Teemu Kerola
22.11.2010 38



Review Questions

- What information PSW needs to contain?
- Why 2-stage pipeline is not very beneficial?
- What elements effect the pipeline?
- What mechanisms can be used to handle branching?
- How does CPU move to interrupt handling?