

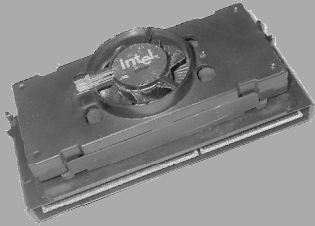
HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI


Lecture 7

CPU Structure and Function

Ch 12.1-4 [Sta06]

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





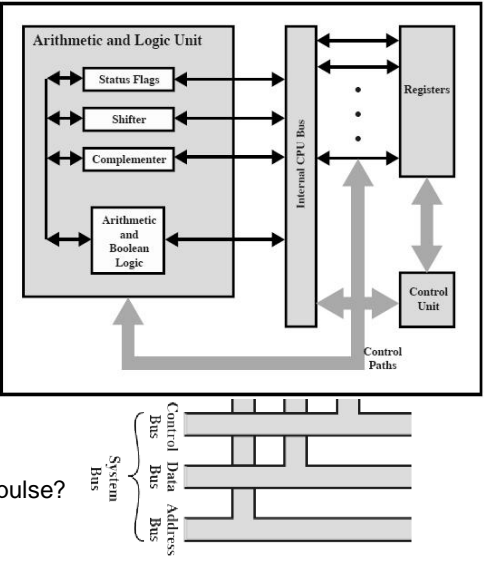
General structure of CPU

(Sta06 Fig 12.2)

- ALU
 - Calculations, comparisons
- Registers
 - Fast work area
- Processor bus
 - Moving bits
- Control Unit (*Ohjausyksikkö*)


(lecture 10, Ch 16-17)

 - What? Where? When?
 - Clock pulse
 - Generate control signals
 - What happens at the next pulse?
- MMU?
- Cache?



Computer Organization II, Spring 2010, Tiina Niklander


9.2.2010 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 (*apurekisterit*) for temporal storage during instruction execution
 - Example: Instruction register (IR) instruction interpretation; operand first to latch and only then to ALU

Computer Organization II, Spring 2010, Tiina Niklander 9.2.2010 3



User visible registers

- Different processor families⇒
 - different number of registers,
 - different naming conventions (*nimeämistavat*),
 - different purposes
- General-purpose registers (*yleisrekisterit*)
- Data registers (*datarekisterit*)
- Address registers (*osoiterekisterit*)
 - Segment registers (*segmenttirekisterit*)
 - Index registers (*indeksirekisterit*)
 - Stack pointer (*pino-osoitin*)
 - Frame pointer (*ympäristöosoitin*)
- Condition code registers (*tilarekisterit*) No condition code regs.
IA-64, MIPS

Computer Organization II, Spring 2010, Tiina Niklander 9.2.2010 4

(Sta06 Fig 12.3)

Example

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/) 16-32 ok! (y 1977)
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, Spring 2010, Tiina Niklander 9.2.2010 5

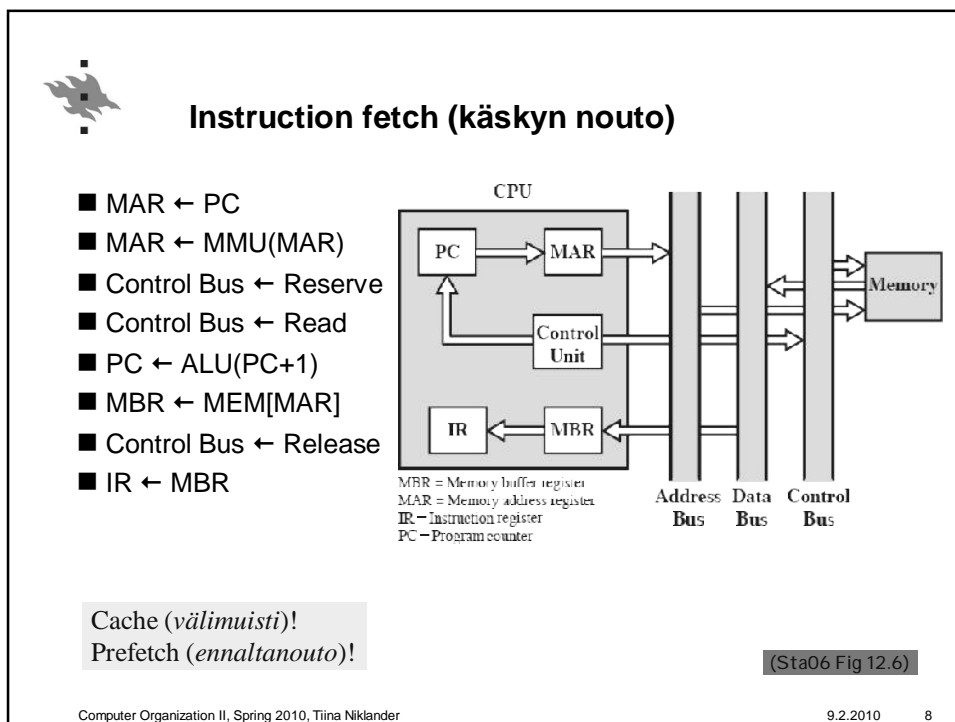
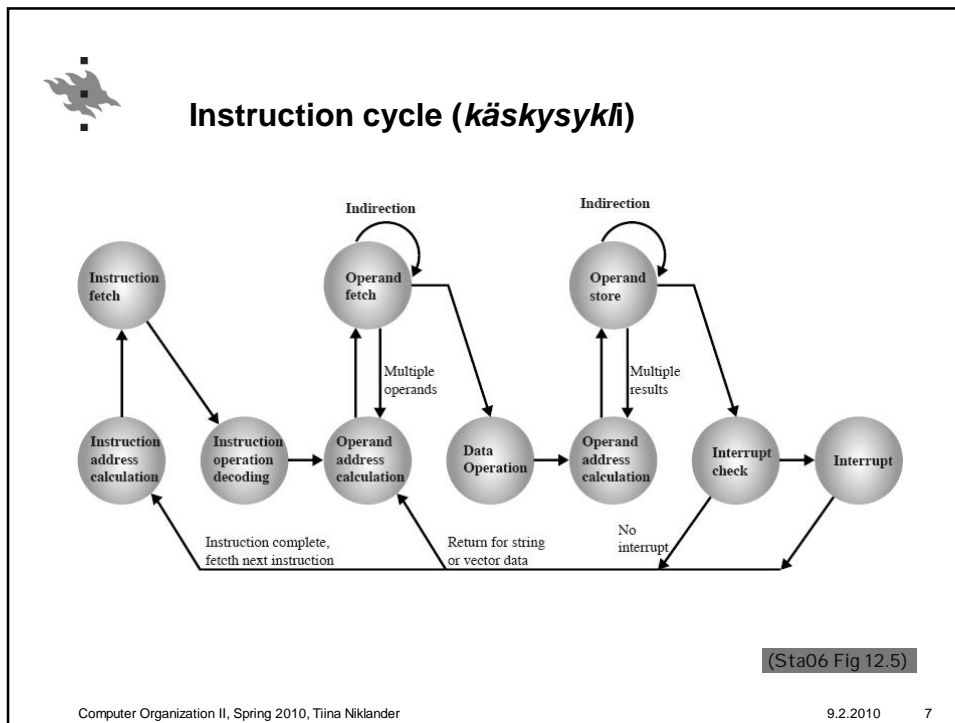
PSW - Program Status Word

- 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, Spring 2010, Tiina Niklander 9.2.2010 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

ALU? Regs? ← MBR

Cache!

(Sta06 Fig 12.7)


Computer Organization II, Spring 2010, Tiina Niklander 9.2.2010 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] ← No address translation!
- Control Bus ← Release

SP = Stack Pointer (Sta06 Fig 12.8)


Computer Organization II, Spring 2010, Tiina Niklander 9.2.2010 10



Computer Organization II


Instruction pipelining (*liukuhihna*)




Computer Organization II, Spring 2010, Tiina Niklander 9.2.2010 11



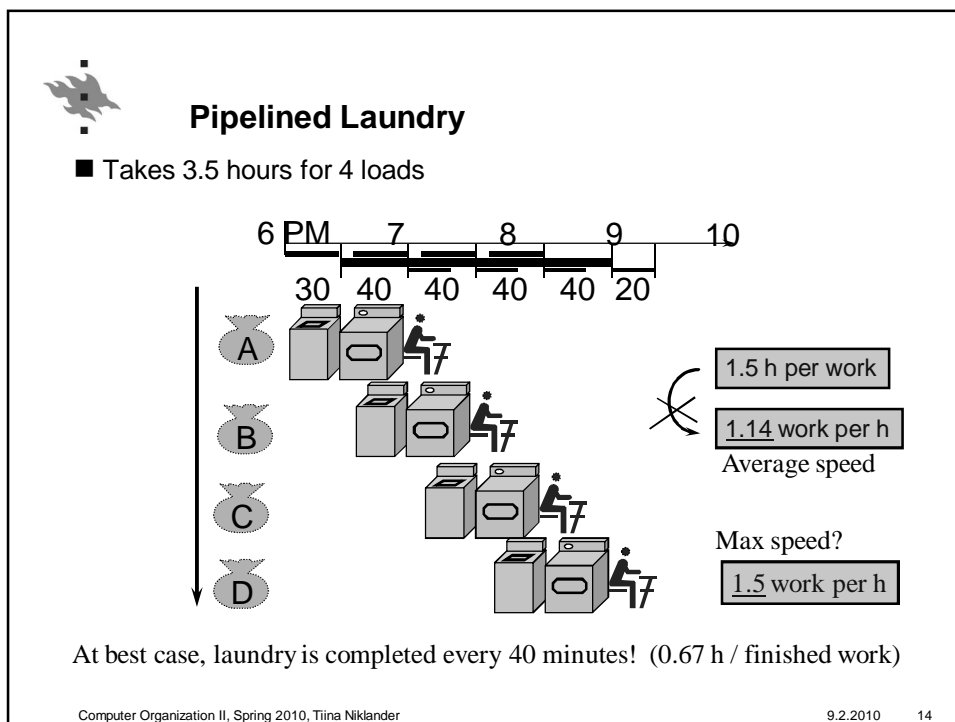
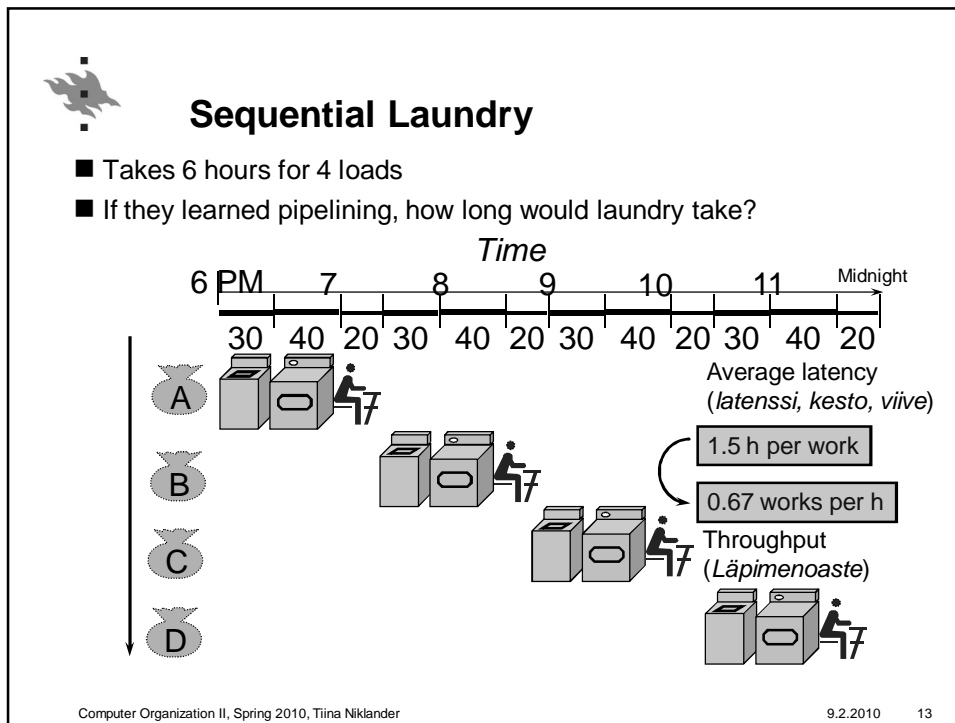
Laundry (*pesula*) example (by David A. Patterson)

- 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, Spring 2010, Tiina Niklander 9.2.2010 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

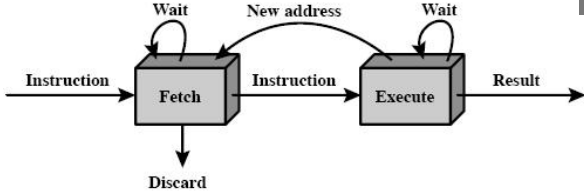
Computer Organization II, Spring 2010, Tiina Niklander
9.2.2010 15

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
- “Hiccups” (*hikka*)
 - Variation in task arrivals, works best with constant flow of tasks

Computer Organization II, Spring 2010, Tiina Niklander
9.2.2010 16

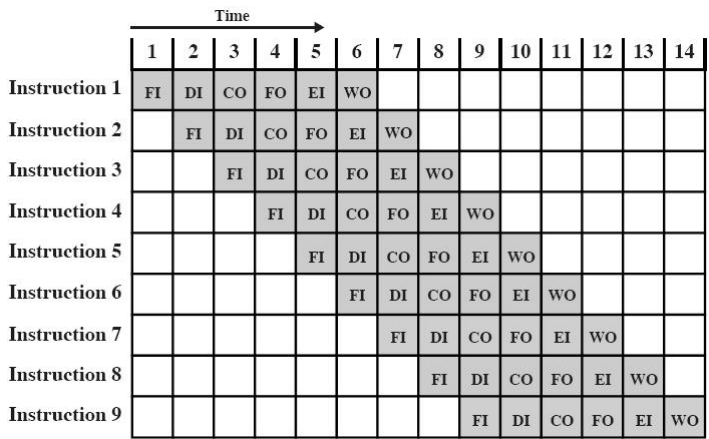
2-stage instruction execution pipeline (2-vaiheinen liukuhinna)



- 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?

Computer Organization II, Spring 2010, Tiina Niklander 9.2.2010 17

6-stage pipeline



FE - Fetch instruction
 DI - Decode instruction
 CO - Calculate operand addresses

FO - Fetch operands
 EI - Execute instruction
 WO - Write operand

(Sta06 Fig 12.10)

Computer Organization II, Spring 2010, Tiina Niklander 9.2.2010 18



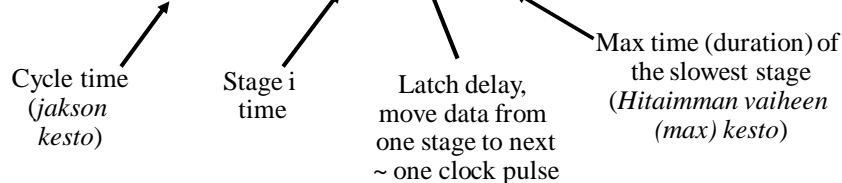
Pipeline speedup (*nopeutus*)?

- Lets calculate (based on Fig 12.10):
 - 6- stage pipeline, 9 instr. → 14 time units
 - Same without pipeline → $9 \cdot 6 = 54$ time units
 - Speedup = $54/14 = 3.86 < 6$!
 - Maximum speedup: one instruction per time unit finish:
9 time units → 9 instructions; $54/9 = 6$
- Not every instruction uses every stage
 - Will not affect the pipeline speed
 - 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)



Pipeline performance: one cycle time

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



- 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

Speedup?

n instructions, k stages, τ = cycle time

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

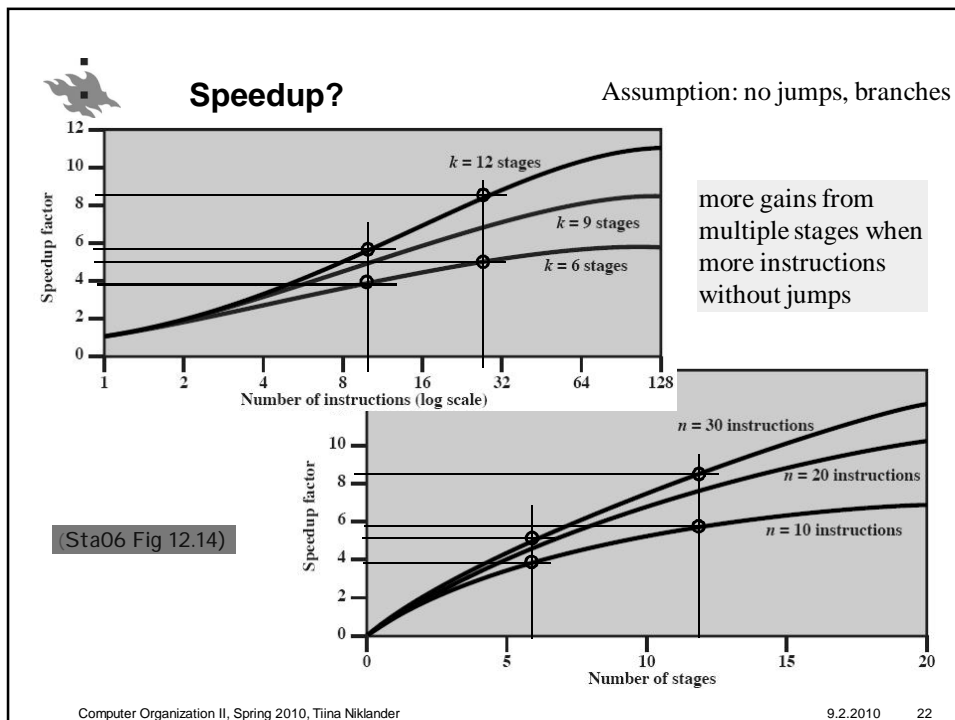
Pipeline: $T_k = [k + (n-1)]\tau$ See Sta06 Fig 12.10 and check yourself!

k stages before the first task (instruction) is finished

next (n-1) tasks (instructions) will finish each during one cycle, one after another

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

Computer Organization II, Spring 2010, Tiina Niklander 9.2.2010 21





More notes

- 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 (*rinnakkainen*) execution of instructions in the pipeline makes them proceed faster as whole, but slows down execution of single instruction



Problems, design issues

- Structural dependency (*rakenteellinen riippuvuus*)
 - Several stages may need the same HW
 - Memory: FI, FO, WO
 - ALU: CO, EI
- Control dependency (*kontrolliriippuvuus*)
 - Jump destination of conditional branch known only after EI-stage
 - → Prefetched wrong instructions
- Data dependency (*datariippuvuus*)
 - Instruction needs the result of the previous non-finished instruction

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

ADD	R1,R7, R9
Jump	There
ADD	R2,R3,R4
MUL	R1,R4,R5

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



Solutions

- Hardware must notice and wait until dependency cleared
 - Add extra waits, “bubbles”, to the pipeline; Commonly used
 - Bubble (*kupla*) delays everything behind it in all stages
- Structural dependency
 - More hardware, f.ex. separate ALUs for CO- and EI-stages
 - Lot 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: result can be accessed already before WO-stage



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

Example: data dependency - RAW

MUL R1, R2, R3

ADD R4, R5, R6

SUB R7, R1, 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		FO	EI	WO		
			FI	DI	CO		FO	EI	WO	

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	FO	EI	WO			
			FI	DI	CO	FO	EI	WO		

Computer Organization II, Spring 2010, Tiina Niklander 9.2.2010 27

Example: Change instruction execution order

MUL R1, R2, R3

ADD R4, R5, R6

SUB R7, R1, R8

ADD R9, R0, 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		FO	EI	WO		
			FI	DI	CO		FO	EI	WO	

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	FO	EI	WO			
			FI	DI	CO	FO	EI	WO		




Computer Organization II, Spring 2010, Tiina Niklander 9.2.2010 28

 **Example: by-pass (oikopolut)**


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			
		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	

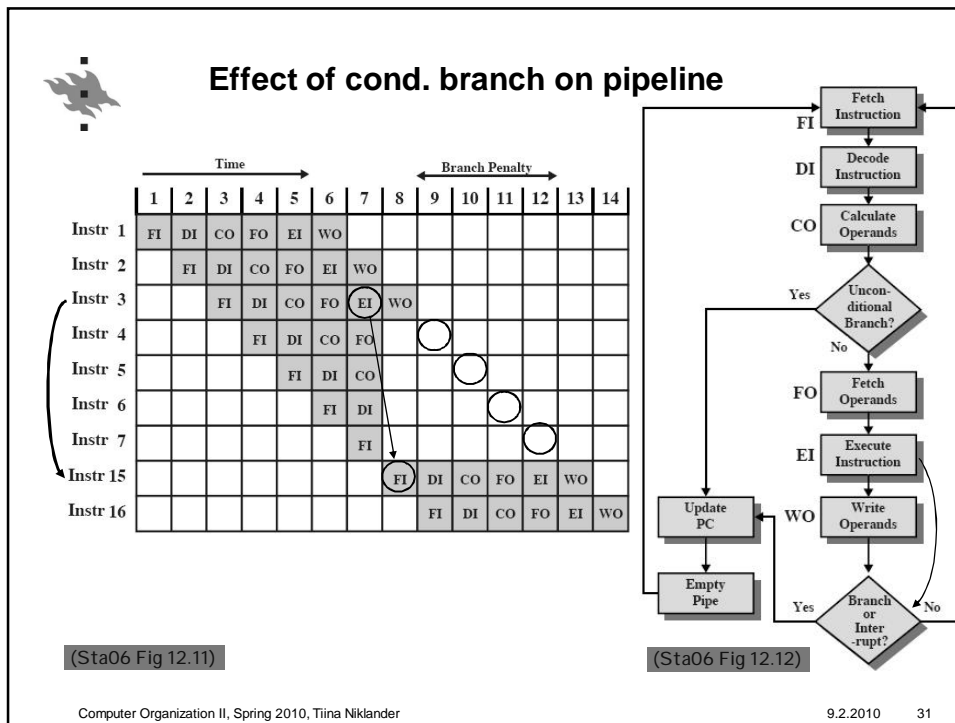
Computer Organization II, Spring 2010, Tiina Niklander 9.2.2010 29

 **Computer Organization II**

Jumps and pipelining (*Hypyt ja liukuhihna*)

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

Computer Organization II, Spring 2010, Tiina Niklander 9.2.2010 30



Delayed branch (*viivästetty haarautuminen*)

- Compiler places some useful instructions (1 or more) after branch instructions;
 - always executed!
 - No roll-back of instructions due incorrect prediction
 - This would be difficult to do
 - If no useful instruction available, compiler uses NOP
- Less actual work lost
 - 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)

```

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

delay slot

Computer Organization II, Spring 2010, Tiina Niklander 9.2.2010 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 other pipeline
 - After branch decision: reject the incorrect pipeline (or results)
- Problems
 - Branch target address known 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& register contention? More ALUs?
- Capability to *cancel* not-taken instruction stream from pipeline

IBM 370/168,
IBM 3033



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



Branch prediction (*hyppyjen ennustus*)

- Make a (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 (reported in LILJ88)

Motorola 68020
VAX 11/780




Branch prediction (*hyppyjen ennustus*)

- Dynamic prediction (*dynaaminen ennustus*)
 - What has happened in the recent history with this instruction
 - Improves the accuracy of the prediction
 - CPU needs internal space for this = branch history table
 - Instruction address
 - Branch target (instruction or address)
 - Decision: taken / not taken

- Simple alternative
 - Predict based on the previous execution
 - 1 bit is enough
 - Loops will always have one or two incorrect predictions

Computer Organization II, Spring 2010, Tiina Niklander 9.2.2010 37

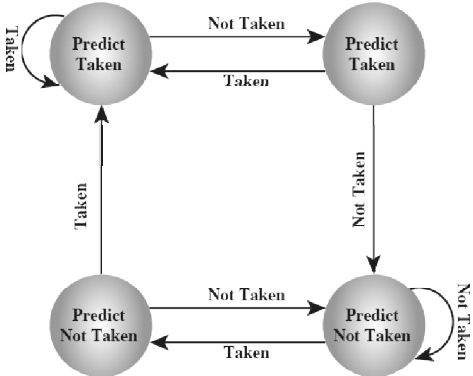


Branch prediction (*hyppyjen ennustus*)

PowerPC 620

- Improved simple model
 - Don't change the prediction so soon
 - Based on two previous instructions
 - 2 bits enough

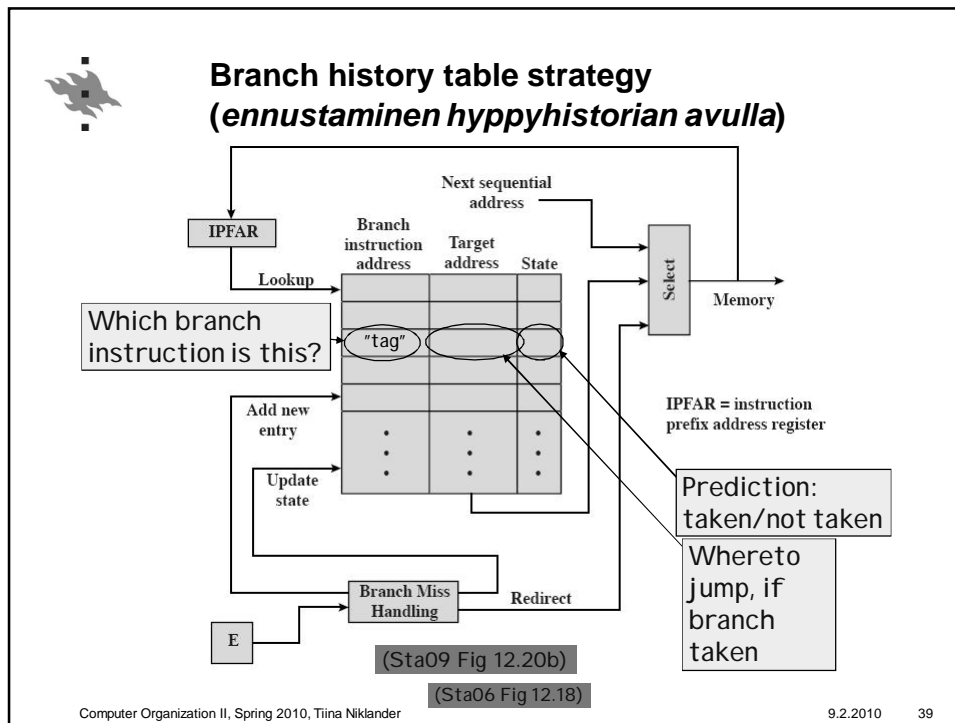
(Sta09 Fig 12.19)
(Sta06 Fig 12.17)



```

            graph TD
              T1((Predict Taken)) -- Taken --> T1
              T1 -- Not Taken --> TN1((Predict Not Taken))
              TN1 -- Taken --> T1
              TN1 -- Not Taken --> TN1
              TN2((Predict Not Taken)) -- Taken --> T2((Predict Taken))
              T2 -- Not Taken --> TN2
              T2 -- Taken --> T2
              T2 -- Not Taken --> TN2
            
```

Computer Organization II, Spring 2010, Tiina Niklander 9.2.2010 38



Review Questions / Kertauskysymyksiä

- 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?

- Mitä tietoja on sisällytettävä PSW:hen?
- Miksi 2-vaiheisesta liukuhihnasta ei ole paljon hyötyä?
- Mitkä tekijät vaikeuttavat liukuhihnan toimintaa?
- Millaisia ratkaisuja on käytetty hyppykäskyjen vaikutuksen eliminoimiseen?
- Kuinka CPU siirtyy keskeytyskäsitteeseen?

Computer Organization II, Spring 2010, Tiina Niklander 40