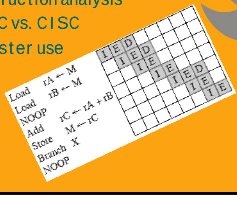
Lecture 8

HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI

CPU Examples & RISC

Ch 12.5-6 [Sta06]
Pentium/ARM

Ch 13 [Sta06]
Instruction analysis
RISC vs. CISC
Register use



Computer Organization II

Pentium

X86: Processor Registers

(Sta09 Table 12.2)

(a) Integer Unit in 32-bit Mode

Type	Number	Length (bits)	Purpose
General	8	32	General-purpose user registers
Segment	6	16	Contain segment selectors
EFLAGS	1	32	Status and control bits
Instruction Pointer	1	32	Instruction pointer

EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI

CS, SS, DS, ES, FS, GS

EFLAGS

EIP

(b) Integer Unit in 64-bit Mode

Type	Number	Length (bits)	Purpose
General	16	32	General-purpose user registers
Segment	6	16	Contain segment selectors
RFLAGS	1	64	Status and control bits
Instruction Pointer	1	64	Instruction pointer

X86: Processor Registers

(Sta09 Table 12.2)

(c) Floating-Point Unit

Type	Number	Length (bits)	Purpose
Numeric	8	80	Hold floating-point numbers
Control	1	16	Control bits
Status	1	16	Status bits
Tag Word	1	16	Specifies contents of numeric registers
Instruction Pointer	1	48	Points to instruction interrupted by exception
Data Pointer	1	48	Points to operand interrupted by exception

Function as a stack, or store MMX values

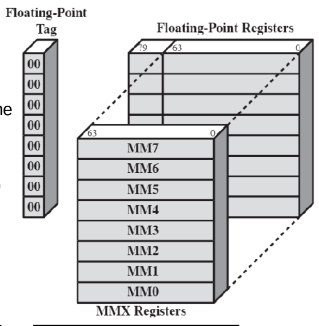
selector, offset

Pentium: FP / MMX Registers

(Sta09 Fig 12.24)

(Sta06 Fig 12.22)

- Aliasing
- FP used as stack (*pino*)
- MMX multimedia instructions use the same registers, but use them with names
- MMX-usage: bits 64-79 are set to 1 → NaN
- FP Tag (word) indicate which usage is current
  - First MMX instr. set
  - EMMS (Empty MMX State) instruction reset

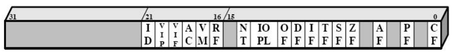


Programmer responsibility

Pentium: EFLAGS Register

(Sta09 Fig 12.22)

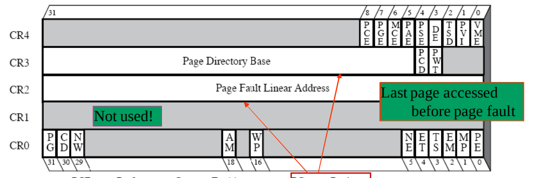
(Sta06 Fig 12.20)



ID =	Identification flag	DF =	Direction flag
VIP =	Virtual interrupt pending	IF =	Interrupt enable flag
VIF =	Virtual interrupt flag	TF =	Trap flag
AC =	Alignment check	SF =	Sign flag
VM =	Virtual 8086 mode	ZF =	Zero flag
RF =	Resume flag	AF =	Auxiliary carry flag
NT =	Nested task flag	PF =	Parity flag
IOPL =	I/O privilege level	CF =	Carry flag
OF =	Overflow flag		

- Condition of the processor: carry, parity, auxiliary, zero, sign, and overflow
  - Used in conditional branches

Pentium: Control Registers



31
8 7 6 5 4 3 2 1 0
CR4
PCE PGE MCE PAE PSE DE TSD PVI VME
CR3
Page Directory Base
PCD PWT
CR2
Page Fault Linear Address
Last page accessed before page fault
CR1
Not used!
CR0
PG CD NW AM WP NE ET TS EM MP PE
31 30 29 18 16 5 4 3 2 1 0

PCE	=	Performance Counter Enable	PG	=	Paging
PGE	=	Page Global Enable	CD	=	Cache Disable
MCE	=	Machine Check Enable	NW	=	Not Write Through
PAE	=	Physical Address Extension	AM	=	Alignment Mask
PSE	=	Page Size Extensions	WP	=	Write Protect
DE	=	Debug Extensions	NE	=	Numeric Error
TSD	=	Time Stamp Disable	ET	=	Extension Type
PVI	=	Protected Mode Virtual Interrupt	TS	=	Task Switched
VME	=	Virtual 8086 Mode Extensions	EM	=	Emulation
PCD	=	Page-level Cache Disable	MP	=	Monitor Coprocessor
PWT	=	Page-level Writes Transparent	PE	=	Protection Enable

System control flags

(Sta09 Fig 12.23)
(Sta06 Fig 12.21)

Pentium: Interrupts

See Sta09 Table 12.3

- Calling interrupt handler; atomic hardware functionality!
  If not in privileged mode (*etuoikeutettu tila*)
  PUSH(SS) stack segment selector to stack
  PUSH(ESP) stack pointer to stack   as subroutine call
  PUSH(EFLAGS) status register to stack
  EFLAGS.IOPL ← 00 set privileged mode
  EFLAGS.IF ← 0 disable interrupts (*keskeytys*)
  EFLAGS.TF ← 0 disable exceptions (*poikkeus*)
  PUSH(CS) code segment selector to stack
  PUSH(EIP) instruction pointer to stack (*käskyosoitin*)
  PUSH(error code) if needed
  number ← interrupt controller / INT-instruction / status register
  CS ← interrupt vector [number].CS
  EIP ← interrupt vector [number].EIP

  Address translation:
  Segment nunber- and
  offset from interrupt vector =>
  Address of the interrupt handler
- Return
  - Privileged IRET-instruction
  - POP everything from stack to their places

Exceptoin and Interrupt Vector Table

Vector Number	Description
0	Divide error; division overflow or division by zero
1	Debug exception; includes various faults and traps related to debugging
2	NMI pin interrupt; signal on NMI pin
3	Breakpoint; caused by INT 3 instruction, which is a 1-byte instruction useful for debugging
4	INTO-detected overflow; occurs when the processor executes INTO with the OF flag set
5	BOUND range exceeded; the BOUND instruction compares a register with boundaries stored in memory and generates an interrupt if the contents of the register is out of bounds.
6	Undefined opcode
7	Device not available; attempt to use ESC or WAIT instruction fails due to lack of external device
8	Double fault; two interrupts occur during the same instruction and cannot be handled serially
9	Reserved
10	Invalid task state segment; segment describing a requested task is not initialized or not valid
11	Segment not present; required segment not present
12	Stack fault; limit of stack segment exceeded or stack segment not present
13	General protection; protection violation that does not cause another exception (e.g., writing to a read-only segment)
14	Page fault
15	Reserved
16	Floating-point error; generated by a floating-point arithmetic instruction
17	Alignment check; access to a word stored at an odd byte address or a doubleword stored at an address not a multiple of 4
18	Machine check; model specific
19-31	Reserved
32-255	User interrupt vectors; provided when INTR signal is activated

Unshaded: exceptions
Shaded: interrupts

Sta09 Table 12.3

Computer Organization II

ARM

ARM features

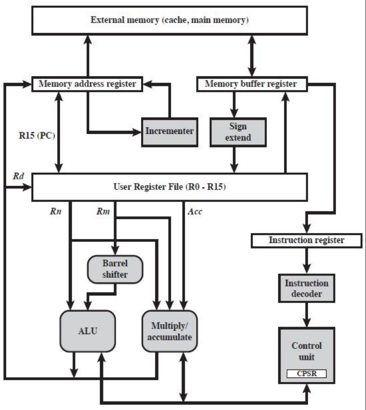
- Array of uniform registers (moderate number)
- Fixed length (32 bit) instruction (Thumb 16 bit)
- Load/Store architecture,
- Small number of addressing modes (reg + instr. field)
- Autoincrement addressing mode (for program loops)
- Data processing instructions allow shift or rotate to preprocess one of source regs
  - Separate ALU and shifter for this purpose
- Conditional execution of instructions
  - Fewer conditional branches, improves pipeline efficiency

ARM Processor Organization

Varies substantially - different versions of ARM architecture

Simplified, generic organization

Register file: set of 32-bit registers, total 37 regs
31 general-purpose regs
6 status regs
Partially overlapping banks



Processor modes

- User mode
  - no access to protected system resources, can cause exception
- Supervisor mode
  - For OS, starts with software interrupt instruction
- Abort mode – due to memory faults
- Undefined mode – instruction not supported
- Fast interrupt mode
  - Interrupt from designated fast interrupt source
  - Not interruptable, can interrupt normal interrupt
- Interrupt mode
  - Any other interrupt signal, can be interrupted by fast interrupt
- System mode
  - Only for certain priviledged OS tasks

Exception modes

Register organization

SP – stack pointer
LR – link register
PC – program counter
CPSR – current program status register
SPSR – saved program status register

Shaded regs replaced in exception modes!

Modes						
	Privileged modes					
		Exception modes				
User	System	Supervisor	Abort	Undefined	Interrupt	Fast Interrupt
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12	R12_fiq
R13 (SP)	R13 (SP)	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14 (LR)	R14 (LR)	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

Program status regs (CPSR & SPSR)

31	30	29	28	27	26 25	24	23 22 21 20	19 18 17 16	15 14 13 12 11 10	9	8	7	6	5	4 3 2 1 0
N	Z	C	V	Q	Res	J	Reserved	GE[3:0]	Reserved	E	A	I	F	T	M[4:0]

User flags | System control flags

- N,Z,C,V – condition code
- Q – overflow or saturation in SIMD-orient. instr.
- J – Jazelle instruction in use
- GE[3:0] – for SIMD as greater than or equal flags for individual bytes or halfwords of the result
- E – endianness in load/store
- A,I,F – interrupt disable bits (A - imprecise data aborts, I – normal IRQ, F – fast FIQ)
- T – normal / Thumb instr.
- M[4:0] – mode

ARM Interrupt vector

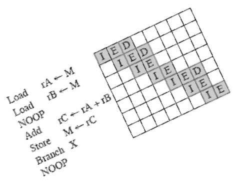
Table lists the exception types and the address in interrupt vector for that type.

The vector contains the start addresses of the interrupt handlers.

Exception type	Mode	Normal entry address	Description
Reset	Supervisor	0x00000000	Occurs when the system is initialized.
Data abort	Abort	0x00000010	Occurs when an invalid memory address has been accessed, such as if there is no physical memory for an address or the correct access permission is lacking.
FIQ (fast interrupt)	FIQ	0x0000001C	Occurs when an external device asserts the FIQ pin on the processor. An interrupt cannot be interrupted except by an FIQ. FIQ is designed to support a data transfer or channel process, and has sufficient private registers to remove the need for register saving in such applications, therefore minimizing the overhead of context switching. A fast interrupt cannot be interrupted.
IRQ (interrupt)	IRQ	0x00000018	Occurs when an external device asserts the IRQ pin on the processor. An interrupt cannot be interrupted except by an FIQ.
Prefetch abort	Abort	0x0000000C	Occurs when an attempt to fetch an instruction results in a memory fault. The exception is raised when the instruction enters the execute stage of the pipeline.
Undefined instructions	Undefined	0x00000004	Occurs when an instruction not in the instruction set reaches the execute stage of the pipeline.
Software interrupt	Supervisor	0x00000008	Generally used to allow user mode programs to call the OS. The user program executes a SWI instruction with an argument that identifies the function the user wishes to perform.

Computer Organization II

RISC-architecture



Ch 13 [Sta06]

- Instructions
- RISC vs. CISC
- Register allocation

Hardware mile stones

- Atlas — Virtual memory, 1962 — Tom Kilburn
  - Simpler memory management
- Atlas — Pipeline, 1962 — Tom Kilburn
- IBM S/360, DEC PDP-8 — Architecture family concept, 1964 — Gene Amdahl
  - Set of computers using the same instruction set
- IBM S/360 — Microprogrammed control, 1964 — Maurice Wilkes
  - Easier control design and impl.
- Univac — Multiple processors, 1964 — J.P. Eckert, John Mauchly
  - test_and_set instruction needed
- IBM S/360 — Cache, 1965 — Maurice Wilkes
  - Huge improvement in performance
- IBM — RISC-architecture, 1980 — John Cocke, 1974; J.L. Hennessy & D.A. Patterson
  - Simple instruction set
- IBM, Intel — Superscalar CPU, 1989 — John Cocke, 1965; IBM; Intel
  - Multiple instruction per cycle
- Intel — Hyperthreading CPU, 2001 — CDC, 1964; Intel
  - Several register sets and virtual processors on chip
- Intel, Sony-Toshiba-IBM — Multicore CPU, 2005 — Intel; IBM
  - Several full processors on chip

CISC (Complex Instruction Set Computer)

- Goal: Shrink the **semantic gap** *(semanttinen kuilu)* between high-level language and machine instruction set
  - Expressiveness of high-level languages has increased
  - "Simple" compilations
    - Language structures match nicely with instructions
  - Lot of different instructions for different purposes
  - Lot of different data types
  - Lot of different addressing modes
  - Complex tasks performed in hardware by control unit, not in the machine code level (single instruction)
    - Less instructions in one program (shorter code)
    - Efficient execution of complex tasks

Operations and Operands, which are used?

- Year 1982, computer: VAX, PDP-11, Motorola 68000
- Dynamic, occurrencies during the execution

	Dynamic Occurrence		Machine-Instruction Weighted		Memory-Reference Weighted	
	Pascal	C	Pascal	C	Pascal	C
ASSIGN	45%	38%	13%	13%	14%	15%
LOOP	5%	3%	42%	32%	33%	26%
CALL	15%	12%	31%	33%	44%	45%
IF	29%	43%	11%	21%	7%	13%
GOTO	—	3%	—	—	—	—
OTHER	6%	1%	3%	1%	2%	1%

Weighted Relative Dynamic Frequency of HLL Operations [PATT82a]

	Pascal	C	Average
Integer Constant	16%	23%	20%
Scalar Variable	58%	53%	55%
Array/Structure	26%	24%	25%

Dynamic Percentage of Operands

80% of references to local variables

(Sta06 Table 13.2, 13.3)

Subroutine (procedure, function) calls?

- Lot of subroutine calls
- Calls rarely have many parameters
- Nested (*sisäkkäinen*) calls are rare

(Sta06 Table 13.4)

Percentage of Executed Procedure Calls With	Compiler, Interpreter, and Typesetter	Small Nonnumeric Programs
>3 arguments	0–7%	0–5%
>5 arguments	0–3%	0%
>8 words of arguments and local scalars	1–20%	0–6%
>12 words of arguments and local scalars	1–6%	0–3%

Procedure Arguments and Local Scalar Variables

- How to use the information?

98% less than 6 parameters
92% less than 6 local variables

Observations

- Most operands are simple
- Many jumps and branches
- Compilers do not always use the complex instructions
  - They use only a subset of the instruction set
- Conclusion?

Occam's razor (Occamin partaveitsi)

"Entia non sunt multiplicanda praeter necessitatem"
("Entities should not be multiplied more than necessary")
William Of Occam (1300-1349)
English monk, philosopher

"It is vain to do with more that which can be done with less"

Optimize

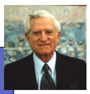
- Optimize the parts that consume most of the time
  - Procedure calls, loops, memory references, addressing, …
- Bad example: rarely used (10%) floating point instructions improved to run 2x:

No speedup   Speedup: 1/2

$$\text{ExTime}_{new} = \text{ExTime}_{old} * (0.9 * 1.0 + 0.1 * 0.5)$$
$$= 0.95 \times \text{ExTime}_{old}$$

$$\text{Speedup} = \text{ExTime}_{old} / \text{ExTime}_{new} = 1 / 0.95 = 1.053 << 2$$

Amdahl's law



Speedup due to an enhancement is proportional to the fraction of the time (in the original system) that the enhancement can be used.

Optimization

- Optimize execution speed (*suoritusnopeus*), instead of of ease of compilation
  - Compilers are good, machines are efficient
    - Compiler can and has time to do the optimization
  - Do most important, common things in hardware and fast
    - E.g. 1-dim array reference
  - And the rest in software
    - E.g. multidim. arrays, string processing, ...
    - Library routines for these

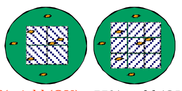
⇨ RISC architecture (Reduced Instruction Set Computer)

RISC architecture

- Plenty of registers (minimum 32)
  - Compilers optimize register usage
- LOAD / STORE architecture
  - Only LOAD and STORE do memory referencing
- Small set of simple instructions
- Simple, fixed-length instruction format (32b)
  - Instruction fetch and decoding simple and efficient
- Small selection of simple address references
  - No indirect memory reference
  - Fast address translation
- Limited set of different operands
  - 32b integers, floating-point
- One or more instructions are done on each cycle

RISC architecture

- CPU easier to implement
  - Pipeline control and optimization simpler
  - Hardwired (*langoitettu*)
- Smaller chip (*piiri*) size
  - More chips per die (*lastu,kiekko*)
  - Smaller waste%
- Cheaper manufacturing
- Faster marketing



25% yield (OK) 55% yield (OK)
75% wasted 45% wasted

RISC vs. CISC

	Complex Instruction Set (CISC)Computer			Reduced Instruction Set (RISC) Computer		Superscalar		
Characteristic	IBM 370/168	VAX 11/780	Intel 80486	SPARC	MIPS R4000	PowerPC	Ultra SPARC	MIPS R10000
Year developed	1973	1978	1989	1987	1991	1993	1996	1996
Number of instructions	208	303	235	69	94	225		
Instruction size (bytes)	2–6	2–57	1–11	4	4	4	4	4
Addressing modes	4	22	11	1	1	2	1	1
Number of general-purpose registers	16	16	8	40 - 520	32	32	40 - 520	32
Control memory size (Kbits)	420	480	246	—	—	—	—	—
Cache size (KBytes)	64	64	8	32	128	16-32	32	64

Characteristics of Some CISCs, RISCs, and Superscalar Processors

(Sta06 Table 13.1)

RISC vs. CISC

Processor	Number of instruction sizes	Max instruction size in bytes	Number of addressing modes	Indirect addressing	Load/store combined with arithmetic	Max number of memory operands	Unaligned addressing allowed	Max Number of MMU uses	Number of bits for integer register specifier	Number of bits for FP register specifier
AMD29000	1	4	1	no	no	1	no	1	8	3 [a]
MIPS R2000	1	4	1	no	no	1	no	1	5	4
SPARC	1	4	2	no	no	1	no	1	5	4
MC88000	1	4	3	no	no	1	no	1	5	4
HP PA	1	4	10 [a]	no	no	1	no	1	5	4
IBM RT/PC	2 [a]	4	1	no	no	1	no	1	4 [a]	3 [a]
IBM RS/6000	1	4	4	no	no	1	yes	1	5	5
Intel i860	1	4	4	no	no	1	no	1	5	4
IBM 3090	4	8	2 [b]	no [b]	yes	2	yes	4	4	2
Intel 80486	12	12	15	no [b]	yes	2	yes	4	3	3
NSC 32016	21	21	23	yes	yes	2	yes	4	3	3
MC68040	11	22	44	yes	yes	2	yes	8	4	3
VAX	56	56	22	yes	yes	6	yes	24	4	0
Clipper	4 [a]	8 [a]	9 [a]	no	no	1	0	2	4 [a]	3 [a]
Intel 80960	2 [a]	8 [a]	9 [a]	no	no	1	yes [a]	—	5	3 [a]

a RISC that does not conform to this characteristic.
b CISC that does not conform to this characteristic.

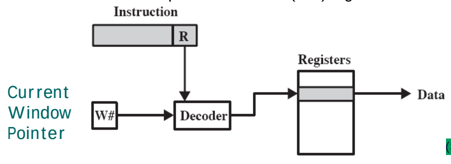
(Sta06 Table 13.7)

Computer Organization II

Register usage

Register storage (Register file)

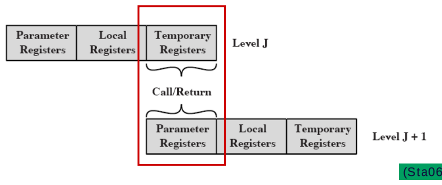
- More registers than addressable in the instruction
  - E.g. SPARC has just 5 bits for register number → 0.. 31, but the processor has 40 to 540 registers
- Small subset of registers available for each instruction in **register window**
  - In the window references to register r0-r31
  - CPU maps them to actual (true) registers r0-r539



(Sta06 Fig 13.3)

Register window (*rekisteri-ikkuna*)

- Procedure call uses registers instead of stack
  - Fixed number of registers for parameters and local variables (*paikalliset muuttujat*)
  - Overlapping area to allow parameter passing to the next procedure and back to caller

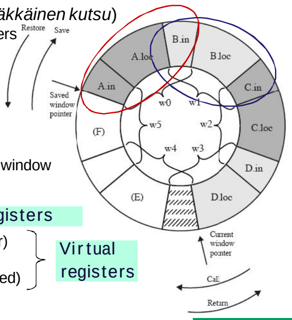


(Sta06 Fig 13.1)

Register window (*rekisteri-ikkuna*)

- Too many nested calls (*sisäkkäinen kutsu*)
  - Most recent calls in registers
  - Older activations saved to memory
  - Restore when nesting depth decreases
  - Overlap only when needed
- Global variable?
  - In memory or own register window
- SPARC
  - r0-r7 global var. Real registers
  - r8-r15 parameters (in caller)
  - r16-r23 local variables
  - r24-r31 parameters (to called)

  Virtual registers



(Sta06 Fig 13.2)

Register set vs. cache

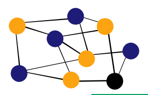
(Sta06 Table 13.5)

Large Register File		Cache
All local scalars	time⇔ space	Recently-used local scalars
Individual variables		Blocks of memory
Compiler-assigned global variables		Recently-used global variables
Save/Restore based on procedure nesting depth		Save/Restore based on cache replacement algorithm
Register addressing	Number of bits	Memory addressing

- The register file acts like a small, fast buffer (as cache?)
  - Register is faster, needs less bits in addressing, **but**
- Difficult for compiler to determine in advance, which of the global variable to place in registers
- Cache decides this issue dynamically
  - Most used and referenced stay in cache

Compiler-based register optimization (allocation of registers)

- Problem: Graph coloring
  - Minimize the number of different colors, while adjacent nodes have different color
- = Difficult problem (NP-compleate)

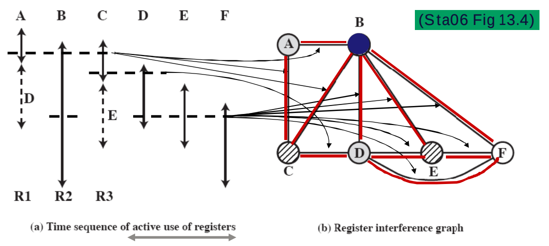


Models of Computation -course

- Form a network of symbolic registers based on the program code
  - Symbolic register~ any program quantity that could be in register
  - The edges of the graph join together program quantities that are used in the same code fragment
- Allocate real registers based on the graph
  - Two symbolic registers that are not used at the same time (no edge between them) can be allocated to the same real register (use the same color)
  - If there are no more free registers, use memory addresses

Allocation of registers (compiler-based register optimization)

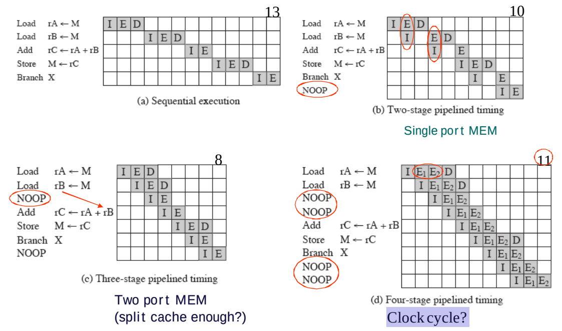
- Node (*solmu*) = symbolic register
- Edge (*särmä*) = symbolic registers used at the same time
- *n* colors = *n* registers



(a) Time sequence of active use of registers (b) Register interference graph

(Sta06 Fig 13.4)

RISC-pipeline



(a) Sequential execution

(b) Two-stage pipelined timing

Single port MEM

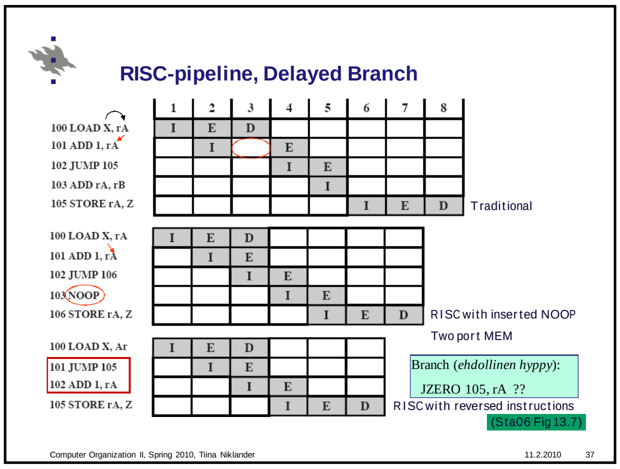
(c) Three-stage pipelined timing

Two port MEM (split cache enough?)

(d) Four-stage pipelined timing

Clock cycle?

(Sta06 Fig 13.6)



RISC-pipeline, Delayed Branch

	1	2	3	4	5	6	7	8
100 LOAD X, rA	I	E	D					
101 ADD 1, rA		I		E				
102 JUMP 105				I	E			
103 ADD rA, rB					I			
105 STORE rA, Z						I	E	D

Traditional

100 LOAD X, rA	I	E	D				
101 ADD 1, rA		I	E				
102 JUMP 106			I	E			
103 NOOP				I	E		
106 STORE rA, Z					I	E	D

RISC with inserted NOOP

Two port MEM

100 LOAD X, Ar	I	E	D			
101 JUMP 105		I	E			
102 ADD 1, rA			I	E		
105 STORE rA, Z				I	E	D

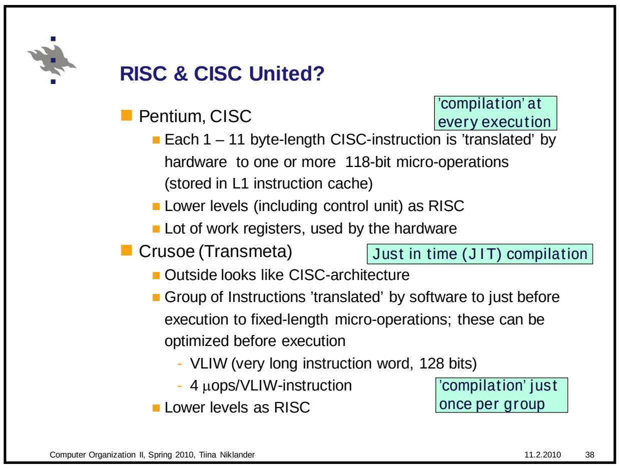
Branch (*ehdollinen hyppy*):

JZERO 105, rA ??

RISC with reversed instructions

(Sta06 Fig 13.7)

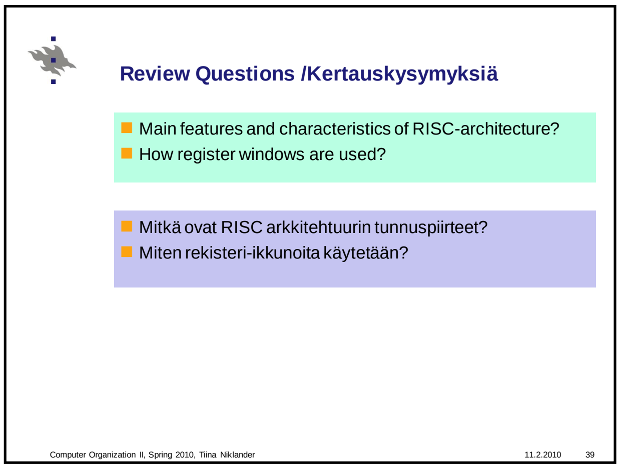
Computer Organization II, Spring 2010, Tiina Niklander 11.2.2010 37



RISC & CISC United?

- Pentium, CISC — 'compilation' at every execution
  - Each 1 – 11 byte-length CISC-instruction is 'translated' by hardware to one or more 118-bit micro-operations (stored in L1 instruction cache)
  - Lower levels (including control unit) as RISC
  - Lot of work registers, used by the hardware
- Crusoe (Transmeta) — Just in time (JIT) compilation
  - Outside looks like CISC-architecture
  - Group of Instructions 'translated' by software to just before execution to fixed-length micro-operations; these can be optimized before execution
    - VLIW (very long instruction word, 128 bits)
    - 4 μops/VLIW-instruction — 'compilation' just once per group
  - Lower levels as RISC

Computer Organization II, Spring 2010, Tiina Niklander 11.2.2010 38



Review Questions /Kertauskysymyksiä

- Main features and characteristics of RISC-architecture?
- How register windows are used?

- Mitkä ovat RISC arkkitehtuurin tunnuspiirteet?
- Miten rekisteri-ikkunoita käytetään?

Computer Organization II, Spring 2010, Tiina Niklander 11.2.2010 39