Prof. Dr.-Ing Jochen H. Schiller
Inst. of Computer Science
Freie Universität Berlin
Germany

TI II: Computer Architecture
ISA and Assembly

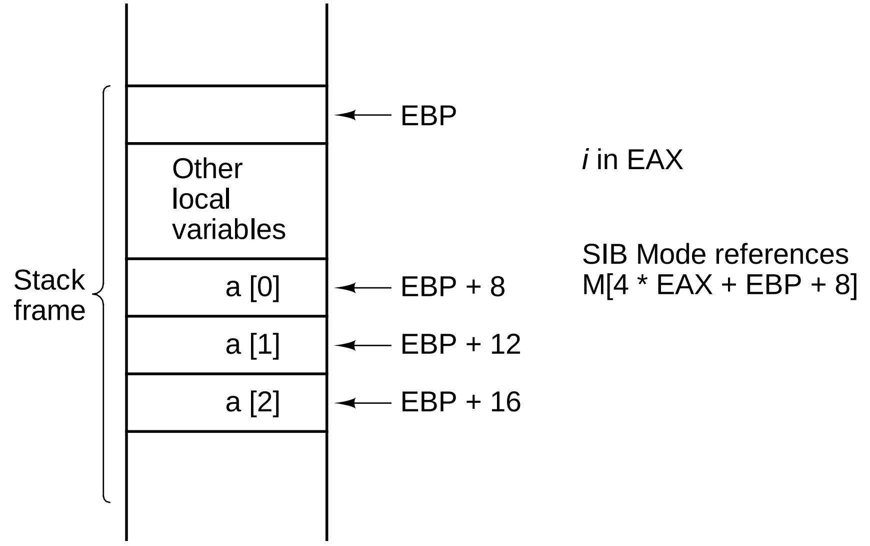
CISC vs. RISC

Data Types

Addressing

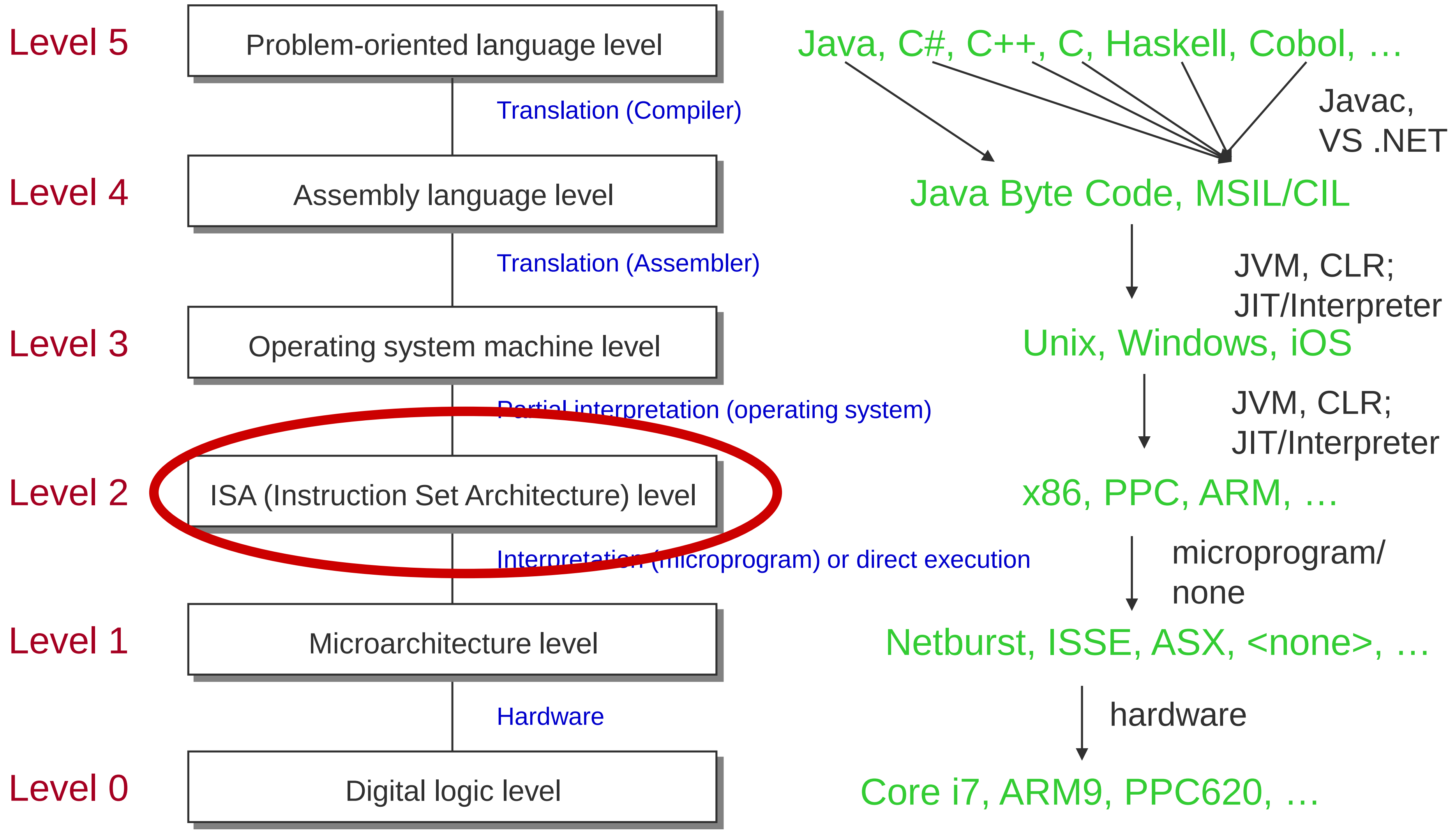
Instructions

Assembler



Content

Where are we now?


Level 5
Problem-oriented language level
Translation (Compiler)
Level 4
Assembly language level
Translation (Assembler)
Level 3
Operating system machine level
Partial interpretation (operating system)
Level 2
ISA (Instruction Set Architecture) level
Interpretation (microprogram) or direct execution
Level 1
Microarchitecture level
Hardware
Level 0
Digital logic level
Java, C#, C++, C, Haskell, Cobol, …
Javac, VS .NET
Java Byte Code, MSIL/CIL
JVM, CLR; JIT/Interpreter
Unix, Windows, iOS
JVM, CLR; JIT/Interpreter
x86, PPC, ARM, …
microprogram/ none
Netburst, ISSE, ASX, <none>, …
hardware
Core i7, ARM9, PPC620, …

Information

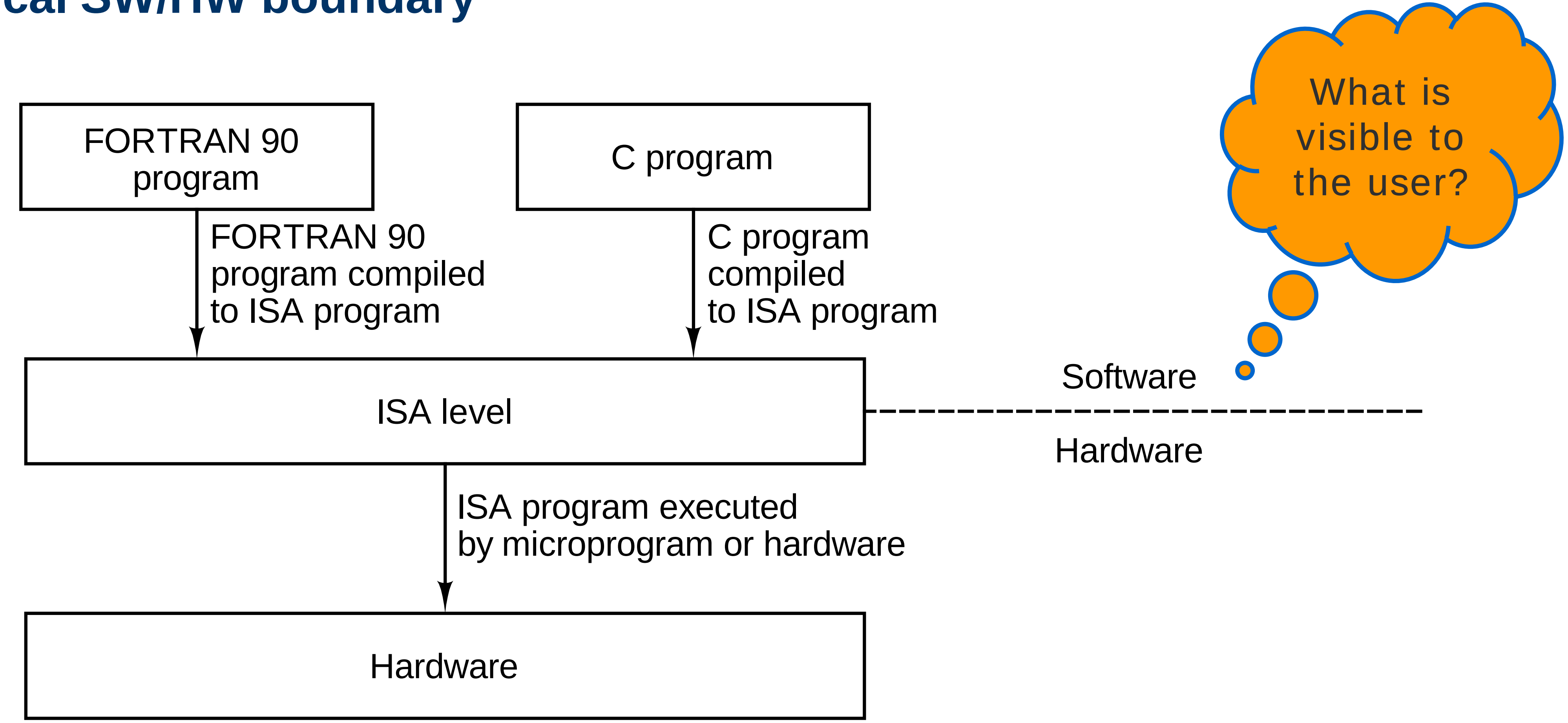
This chapter is mainly for some background information about ISAs, assembler etc.

The assignments plus tutorials teach you how to program with assembler

We skip the operation system part as this is covered in the course “Operating Systems and Computer Networks”

- Subsystems, Interrupts and System Calls
- Processes
- Memory
- Scheduling
- I/O and File System
- Booting, Services, and Security

The classical SW/HW boundary


FORTRAN 90 program
C program
FORTRAN 90 program compiled to ISA program
C program compiled to ISA program
ISA level
Software
Hardware
ISA program executed by microprogram or hardware
Hardware
What is visible to the user?

Execution of operations

Interpretation or compilation of instructions from the ISA-layer

- High-level languages translated into instructions from the ISA
- Pure RISC processors can directly execute ISA instructions
  - i.e. the hardware (HW) can execute these instructions
- More complex processors use **microprogramming**, flexibility and compatibility reasons:
  - hardware changes leave ISA unchanged
  - reprogramming to circumvent hardware problems
  - more powerful ISA – simpler compilers

Example: A selection of the Pentium II integer instructions

Moves

MOV DST,SRC	Move SRC to DST
PUSH SRC	Push SRC onto the stack
POP DST	Pop a word from the stack to DST
XCHG DS1,DS2	Exchange DS1 and DS2
LEA DST,SRC	Load effective addr of SRC into DST
CMOV DST,SRC	Conditional move

Arithmetic

ADD DST,SRC	Add SRC to DST
SUB DST,SRC	Subtract DST from SRC
MUL SRC	Multiply EAX by SRC (unsigned)
IMUL SRC	Multiply EAX by SRC (signed)
DIV SRC	Divide EDX:EAX by SRC (unsigned)
IDIV SRC	Divide EDX:EAX by SRC (signed)
ADC DST,SRC	Add SRC to DST, then add carry bit
SBB DST,SRC	Subtract DST & carry from SRC
INC DST	Add 1 to DST
DEC DST	Subtract 1 from DST
NEG DST	Negate DST (subtract it from 0)

Binary coded decimal

DAA	Decimal adjust
DAS	Decimal adjust for subtraction
AAA	ASCII adjust for addition
AAS	ASCII adjust for subtraction
AAM	ASCII adjust for multiplication
AAD	ASCII adjust for division

Transfer of control

JMP ADDR	Jump to ADDR
Jxx ADDR	Conditional jumps based on flags
CALL ADDR	Call procedure at ADDR
RET	Return from procedure
IRET	Return from interrupt
LOOPxx	Loop until condition met
INT ADDR	Initiate a software interrupt
INTO	Interrupt if overflow bit is set

Boolean

AND DST,SRC	Boolean AND SRC into DST
OR DST,SRC	Boolean OR SRC into DST
XOR DST,SRC	Boolean Exclusive OR SRC to DST
NOT DST	Replace DST with 1 s complement

Shift/rotate

SAL/SAR DST,#	Shift DST left/right # bits
SHL/SHR DST,#	Logical shift DST left/right # bits
ROL/ROR DST,#	Rotate DST left/right # bits
RCL/RCR DST,#	Rotate DST through carry # bits

Test/compare

TST SRC1,SRC2	Boolean AND operands, set flags
CMP SRC1,SRC2	Set flags based on SRC1 - SRC2

Strings

LODS	Load string
STOS	Store string
MOVS	Move string
CMPS	Compare two strings
SCAS	Scan Strings

Condition codes

STC	Set carry bit in EFLAGS register
CLC	Clear carry bit in EFLAGS register
CMC	Complement carry bit in EFLAGS
STD	Set direction bit in EFLAGS register
CLD	Clear direction bit in EFLAGS reg
STI	Set interrupt bit in EFLAGS register
CLI	Clear interrupt bit in EFLAGS reg
PUSHFD	Push EFLAGS register onto stack
POPFD	Pop EFLAGS register from stack
LAHF	Load AH from EFLAGS register
SAHF	Store AH in EFLAGS register

Miscellaneous

SWAP DST	Change endianness of DST
CWQ	Extend EAX to EDX:EAX for division
CWDE	Extend 16-bit number in AX to EAX
ENTER SIZE,LV	Create stack frame with SIZE bytes
LEAVE	Undo stack frame built by ENTER
NOP	No operation
HLT	Halt
IN AL,PORT	Input a byte from PORT to AL
OUT PORT,AL	Output a byte from AL to PORT
WAIT	Wait for an interrupt
SRC = source	# = shift/rotate count
DST = destination	LV = # locals

Some call it “complete”…

COMPLEX INSTRUCTION SET COMPUTER (CISC)

Complex Instruction Set Computer (CISC) 1

Reasons for CISC

- Execution of complex instructions faster than execution of equivalent programs with the same function
- Micro programming allows for more complex instructions
- More complex instructions lead to shorter programs thus faster loading (transfer-rate gap between CPU internally and CPU-main memory)
- Bigger is better – more instructions sound more powerful…it’s marketing!
- Direct support of programming constructs of higher languages using more complex instructions (e.g. string compare)
- Support of specialized powerful compilers
- Compatibility (we can do everything like before plus xyz)
- Support of special purpose applications (e.g. matrix operations)

➔ more transistors/chip, higher programming languages and special purpose applications favor “complex” instructions

Complex Instruction Set Computer (CISC) 2

Reasons against CISC

- Much faster main memories (argument of the 80’s, today again a problem) and the use of cache memory speed-up program execution
- Micro programs are more and more complex (so where is the difference between programming and micro programming…)
- Replacement of complex instructions using several simpler (much faster) instructions
- Longer development cycles
- Very complex control units
- Large micro programs with (potentially with errors)
- Real programs use only a small fraction of the large instruction set frequently!

CISC – really needed?

System programs in XPL on IBM/360:

- 90% of all instructions used: 10 different instructions
- 95% of all instructions used: 21 different instructions
- 99% of all instructions used: 30 different instructions

COBOL programs on IBM/370:

- 90% of all instructions used: 26 different instructions
- 99% of all instructions used: 48 different instructions
- Only 84 different instructions used at all

The 10 most used instructions in SPECint92 for Intel x86

Instruction	Percentage [%]
load	22
conditional branch	20
compare	16
store	12
add	8
and	6
sub	5
move register-register	4
call	1
return	1
Total	**95**

Limitations of CISC architectures

Usage of instructions (80/20 rule)

- Only 20% of the instructions used frequently
- Many powerful instructions (rarely used)
- Complex instruction format(s)
- Micro programming

Critical problem: number of cycles per instruction (CPI)

- Many classical CISC architectures have CPI >> 2
  - Motorola MC68030: CPI = 4-6
  - Intel 80386: CPI = 4-5
- BUT: optimized code for Pentium/Itanium/… – typical CPI ≈ 1
  - Superscalar processors e.g. issuing 4 instructions in parallel could theoretically go down to 0.25, but: floating-point, SIMD, branch mis-predictions, memory latency …

REDUCED INSTRUCTION SET COMPUTER (RISC)

Reduced Instruction Set Computer (RISC)

The instruction set consists of

- a few, absolutely necessary instructions (≤ 128) and
- instruction formats (≤ 4) with a
- fixed instruction length of 32 bit and only some
- addressing modes (≤ 4).

This allows a much simpler implementation of the control unit and saves space on the chip for additional units.

Many general-purpose registers, at least 32, are needed.

Memory access is only possible via special load and store instructions.

Register/register architecture

Memory access is via load and store operations only.

All other instructions work on the CPU registers only, e.g., arithmetic operations load operands from registers and store results in registers only.

This basic principle is called

- register/register architecture or
- load/store architecture and is typical for many (original) RISC computers.

Additional features of RISC computers

If possible, all instructions should be implemented in a way that they finish within a single processor cycle.

Consequence: pure RISC processors do not use micro programming

- RISC processors introduced enhanced pipelining mechanisms (today, many processors use pipelining for the micro instructions, e.g., Pentium 4 and up).

Furthermore, the early RISC processors had a software-controlled pipeline (compilers inserted delay NOPs, introduced delayed jumps etc.) instead of special hardware.

Aside

- PC processors like the Pentium 4 (and up) use micro programming, the internal micro architecture (netburst) is rather RISC, the ISA is CISC.

RISC

Reasons for

- Single-chip implementation (yes, today “everything” fits on a single chip)
- Shorter development cycles
- Higher clock rates, pipelining
- Re-use of saved chip space for, e.g., cache

Reasons against

- Bottleneck in the memory interface, today again main memory is much slower compared to internal registers/cache
- Space on a chip is not that critical anymore

Early RISC processors

IBM 801 project

- Already 1975, Cocke, IBM research, Yorktown Heights

MIPS project

- Started 1981, Hennessy at the University of Stanford
- The first fully functioning chip was finished in 1983 (NMOS VLSI)
- This project was the starting point of the MIPS corporation.

Berkeley RISC project

- Started 1980, Patterson at UC Berkeley
- Origin of the SPARC processor
- Basic principle of overlapping register windows
- The instruction set contained only 31 instructions with a fixed length of 32 bit and only 2 instruction formats
- Only 3 addressing modes

RISC from today's perspective

What is left from the early ideas of RISC (in many controllers, RISC processors, RISC cores), e.g.:

- Instruction pipelining
- Load/Store architecture
- Large register file, e.g.,
  - 32 general-purpose and
  - 32 floating point register
- A unified instruction format, e.g., 32 bit
- Few addressing modes
- No micro programming

- Good example: RISC-V, https://en.wikipedia.org/wiki/RISC-V - RISC with some extras
  - SIMD/vector processing
  - Hypervisor/virtualization support
  - Different versions depending on use (embedded, 32/64/128 bit, …)
  - Compressed instructions
  - …

Differences between RISC and CISC processors 1

Pure RISC prefer the Harvard architecture

- Separate memory for instructions and data (operands) and, thus, two address and two data busses

➡ parallel fetching of instruction(s) and operand(s) possible

Simplified versions

1. Two separate bus systems up to the L1 caches, but only one main memory/unified L2/L3 cache (cheaper, standard with today’s systems)
2. Only a single, multiplexed bus system

Differences between RISC and CISC processors 2

Control unit

- Hard-wired
- Instruction register is a simple FIFO queue
- Each pipeline stage has its own register
- A simple combinational circuit can “interpret” the OpCodes in each stage directly

Register file

- Consists of a large number of (general purpose) registers
- Supports the simultaneous selection of several registers
    - E.g. 4 port register file: simultaneous write in R0, R1 and read from R2, R3

Differences between RISC and CISC processors 3

Execution unit

- Uses a load/store architecture, loads the operands in parallel via 2 operand busses from the register file and writes back the result within the same clock cycle into the register file.
- No direct connection between ALU and external bus, all data transfer done via load/store via the register file.

Exception

- Register bypass to avoid pipeline hazards (forwarding techniques)

The future of RISC?

Today

- Again, processors much faster than RAM/interconnection
- Frequent load/stores as bottleneck
- Integration of > 5 billion transistors on a single chip feasible

Thus

- Development of VLIW (Very Large Instruction Word) processors
- HP/Intel Itanium, very short pipeline, compiler does most of the work, powerful ISA (IA-64), less memory accesses
- Commercial Failure! But still some use: https://en.wikipedia.org/wiki/Very_long_instruction_word

The future?

- RISC considered harmful? Not in embedded systems…
- Will legacy stay there forever…
    - seems so with x86-64 and similar…
- New opportunities with open architectures like RISC-V

Questions & Tasks

- Check the instruction set of current processors (Intel, ARM, AMD, …) – is it RISC or CISC?
- Which of the initial RISC ideas survived?
- There is a trade-off between RISC and CISC – when to favor RISC? Why using CISC?

Examples of ISAs

CISC, RISC, VM & ADDRESSING

Typical processors: Pentium, SPARC, JVM

The following processors serve as examples for different ISAs

Pentium, Core i3/5/7/9

- Originates from the classical x86 CISC architectures, today 64 bit
- Still CISC to the outside, but many RISC features inside
- Other CISC examples: Athlon, …, Ryzen, many classical ancestors (VAX, IBM, ...)

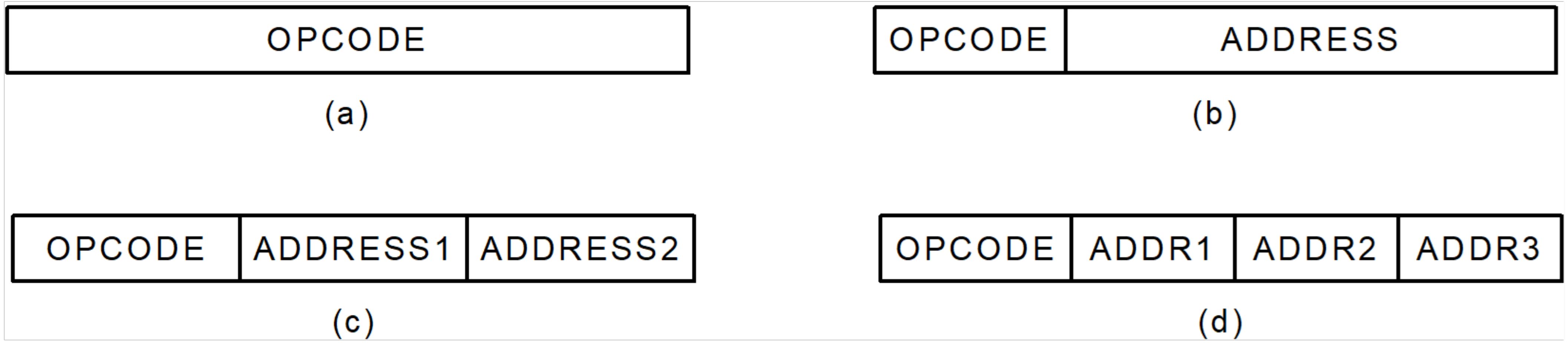
UltraSPARC (Ultra Scalable Processor Architecture)

- Originates from the early RISC projects (like the MIPS processor)
- Still RISC, although extended in many ways
- Can be found in, e.g., SUN computers, industry control systems
- Other RISC examples: Alpha, MIPS, Power, PowerPC, RISC-V

JVM (Java Virtual Machine)

- Either seen as virtual processor or real HW (e.g., picoJava)
- Stack machine (operations take place on a stack)
- Heavily biased by Java
- Other virtual machine examples: CLR, P-Code, WebAssembly

Instruction formats



Example: C:= A + B

a) Zero-address instruction
   - stack architectures: **`push A; push B; ADD; pop C`**

b) One-address instruction
   - Accumulator implicitly operand and result: **`load A; ADD B; st C`**

c) Two-address instruction
   - One operand becomes result: **`ADD B,A; move A,C`**

d) Three-address format
   - **`ADD C,A,B`**

Addressing modes

Addressing mode

- Different possibilities to calculate the address of an operand or the branch target address in the memory

Classical

- Address of operands or branch target address directly given in the instruction (absolute address)

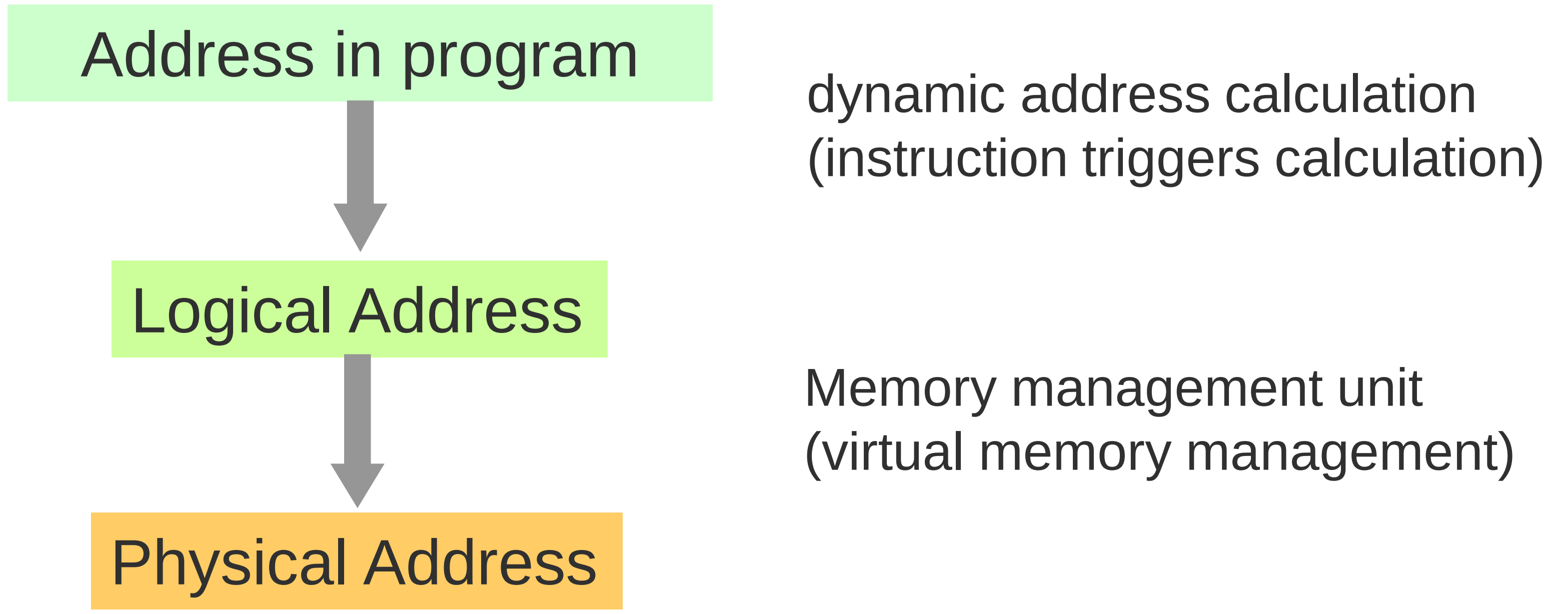
Disadvantages

- Absolute addresses are fixed during programming and, thus, the compiled program determines its location in memory
- Accessing, e.g., dynamic tables require a change in the absolute address for each instruction – ROMs cannot be used as instruction memory!

Addressing modes

Solution

- Calculation of the address during runtime (dynamic address calculation)


Address in program
dynamic address calculation
(instruction triggers calculation)
Logical Address
Memory management unit
(virtual memory management)
Physical Address

Addressing modes – some examples

Be aware:

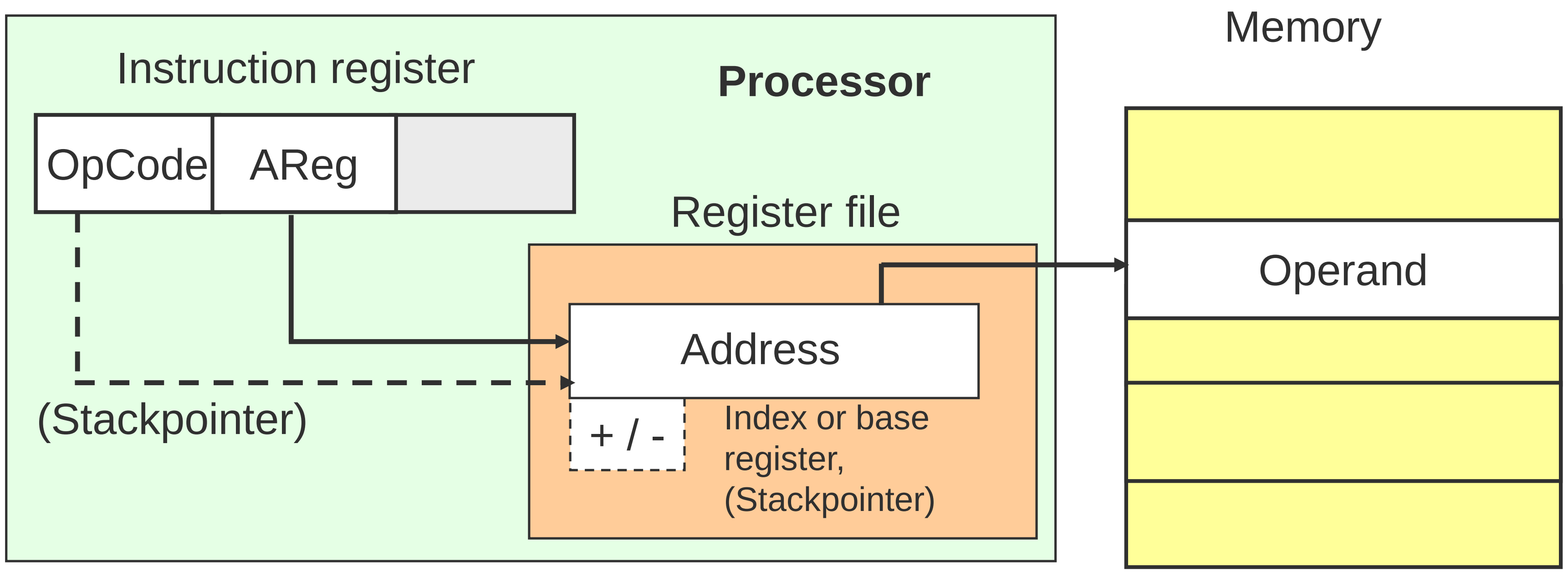
- Naming may differ depending on the architecture
- Not all processors support all modes

Absolute/direct:	`jmpa address`	➔	`PC := address`
PC relative:	`jmpo offset`	➔	`PC := PC’ + offset`
Register indirect:	`jmpr R`	➔	`PC := R`
Sequential execution:	`nop`	➔	`PC := PC’`
Register (direct):	`mul R1,R2,R3`	➔	`PC := PC’; R1 := R2*R3`
Base plus offset:	`load R1,R2,val`	➔	`PC := PC’; R1 := mem(R2 + val)`
Immediate:	`add R1,R2,val`	➔	`PC := PC’; R1 := R2 + val`
Implicit:	load x	➔	`PC := PC’; accumulator := x`

Indexed absolute, base plus index (plus offset), scaled, autoincrement/-decrement, …

- See https://en.wikipedia.org/wiki/Addressing_mode

Example: register indirect addressing



Example:

LD R1, (A0) *(load)*

(Load the register R1 with the content of the memory word the address register A0 points to)

Questions & Tasks

- Several or more complex addressing modes vs. load/store architecture – advantages/disadvantages?
- What is the advantage of using a virtual machine such as JVM, CLR, etc.?
- What is the motivation behind relative / logical or virtual / physical addresses?

PROCEDURES, TRAPS, INTERRUPTS & CO.

Procedures, Traps, Interrupts & Co.

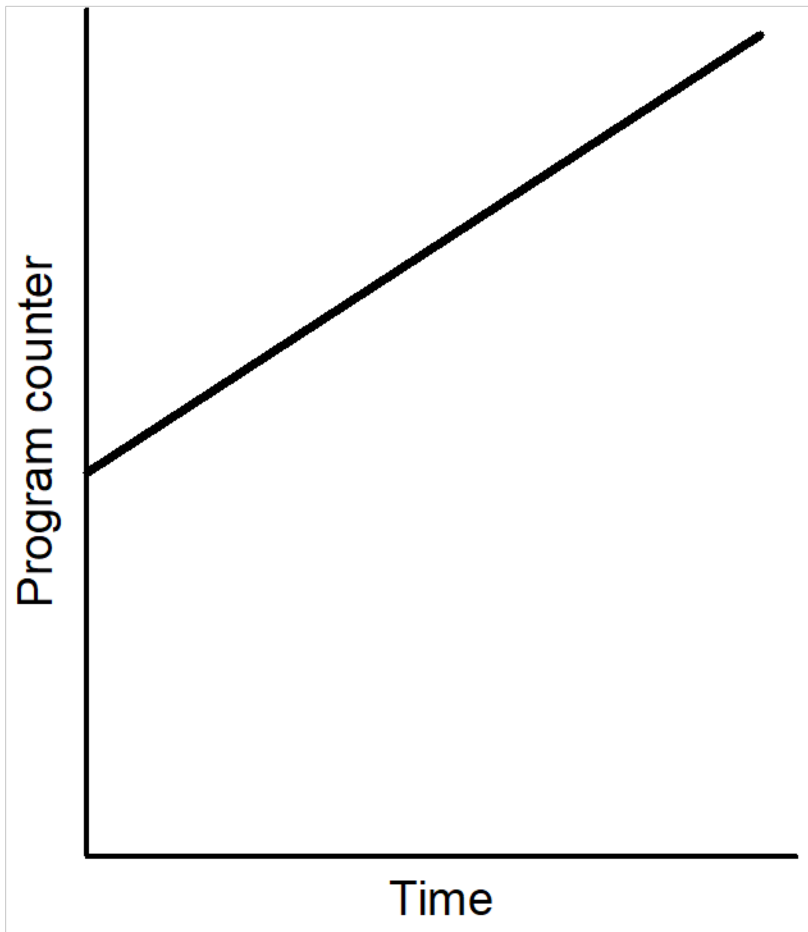
Many reasons for non-linear program execution

- Jumps, branches
- Procedure calls, subroutines, method invocation
- Multithreading, parallel processes, co-routines
- Hardware interrupts (processor external reasons)
- Traps, software interrupts (processor internal reasons)

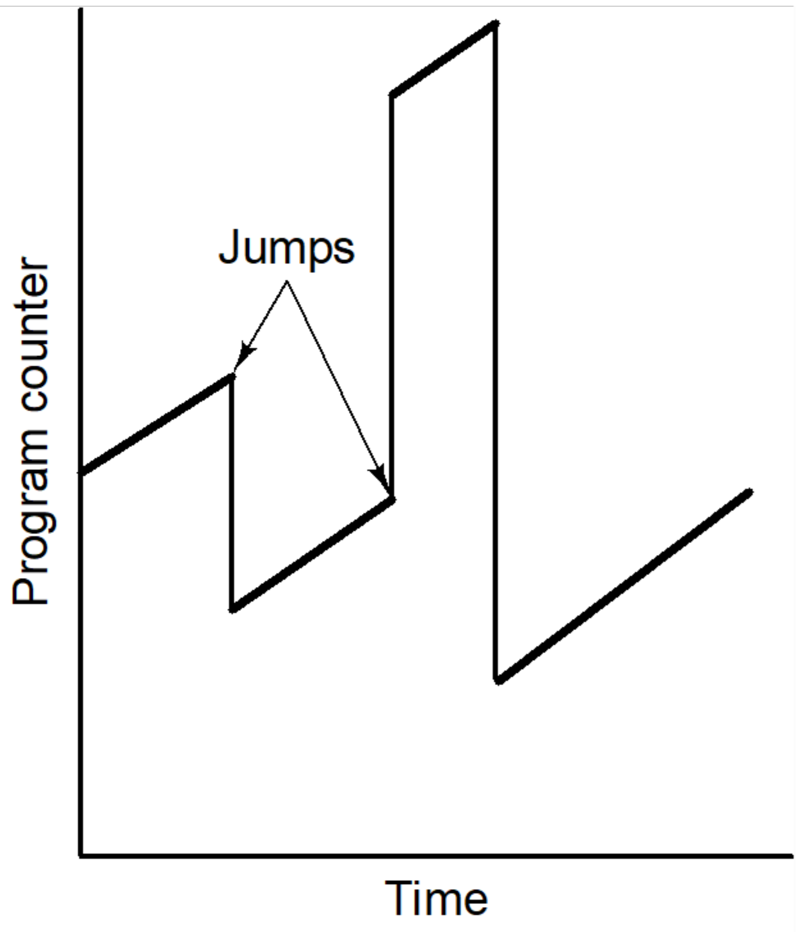
Non linear program execution is the normal case!

- And invalidates standard cache content ...
    - Trace caches can help (more later)

Program execution

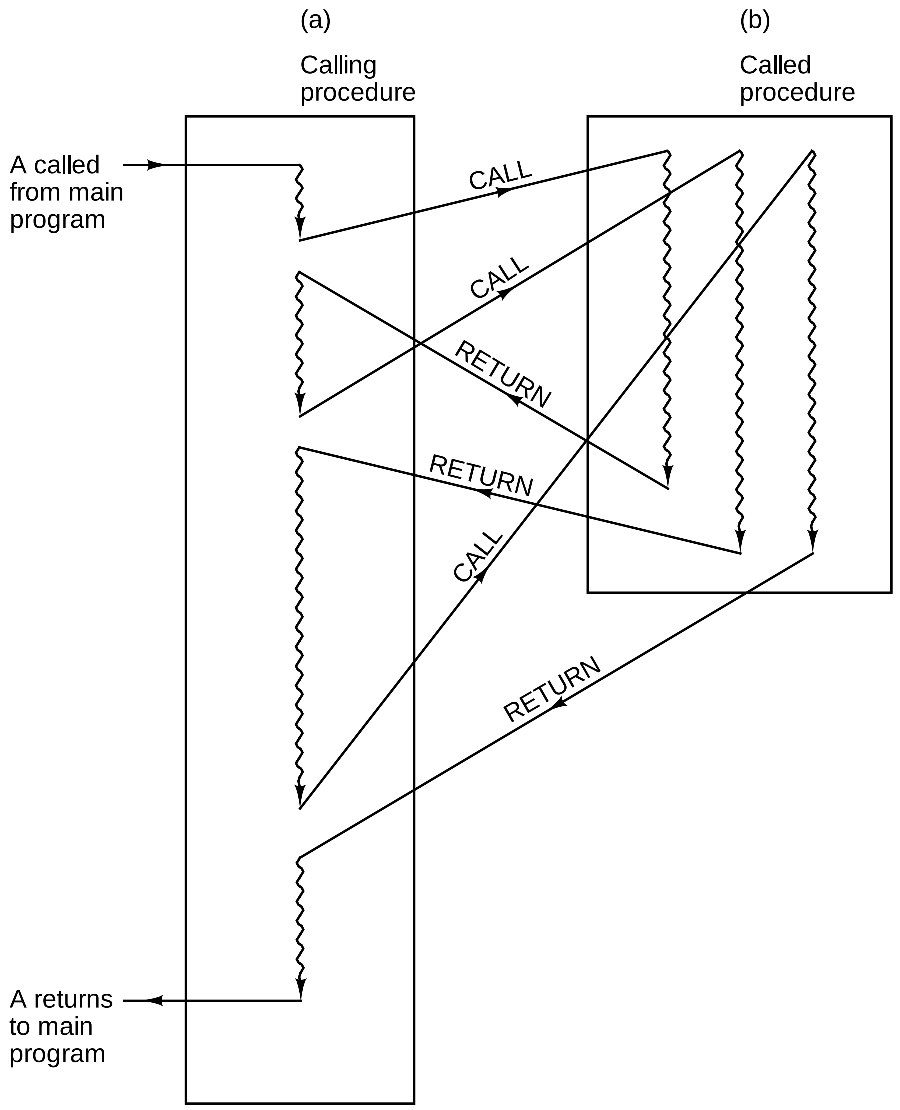


Linear, without branches

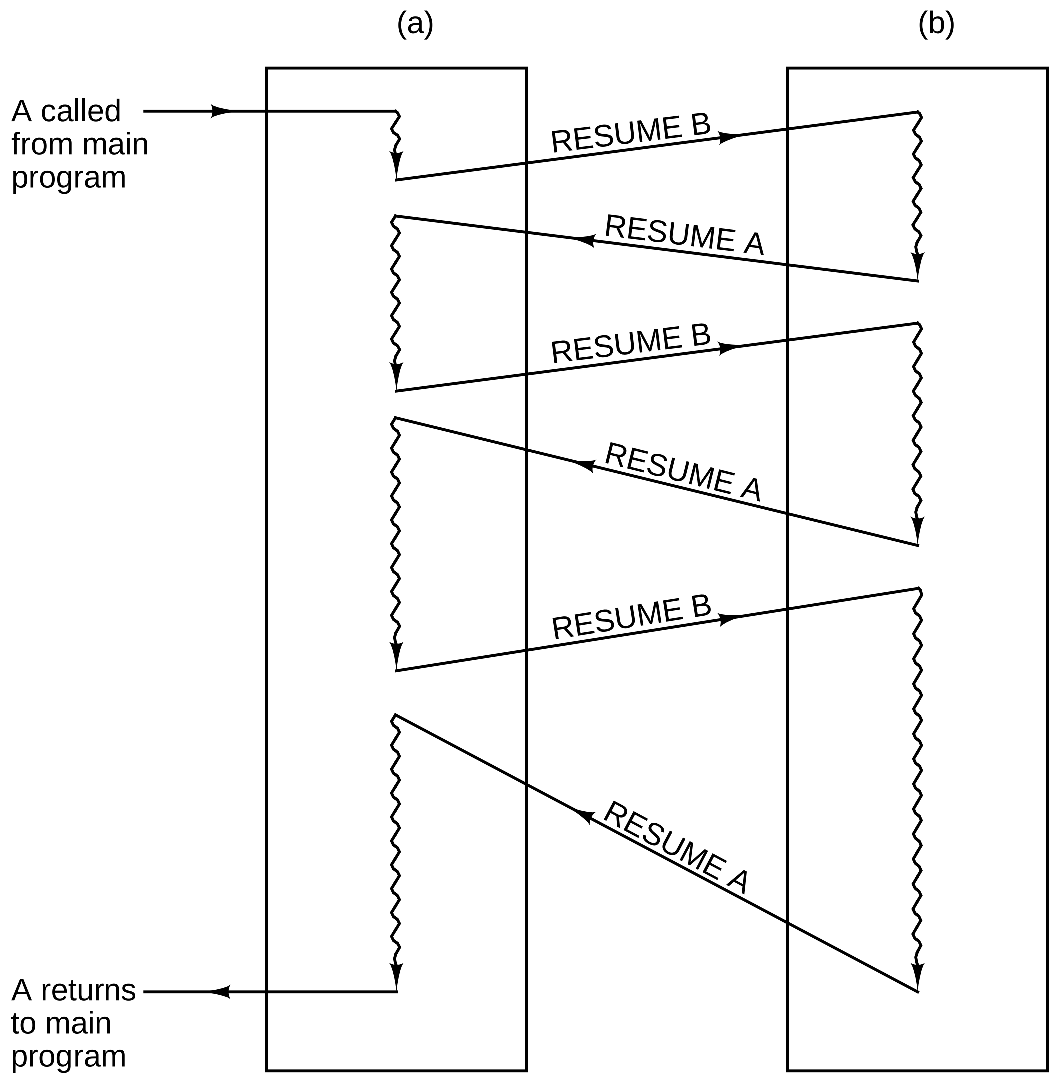


With branches

Procedure call (subroutine, method, ...)



Co-routine call (parallel process, multithreading,...)



How to handle exceptions?

During runtime exceptions may occur, i.e., interruptions of the programmed flow of instructions

Reasons

- Errors in the operating system while executing application programs or errors in the hardware
- Requests of external components for attention of the processor
- …

Exceptional situations may require the interruption of the currently running program or even its termination

Are exceptions exceptional?

Exception handling

Handling of exceptions requires specialized routines (Interrupt Service Routine, ISR)

A specialized hardware component (interrupt system, interrupt controller) typically supports the selection and activation of an ISR

An ISR has the same structure as a subprogram, but there are also some differences

ISR vs. subprogram/subroutine

Activity	**Subroutine/subprogram**	**ISR**
Activation	`call` *subroutine*	`INT` instruction or hardware activation
Return after completion	`RET` instruction *(return from subroutine)*	`RETI` instruction *(return from interrupt)*
Calculation of starting address	Starting address of called subroutine written in calling program	Starting address of called ISR determined via interrupt table
Saving status	Subroutine call typically saves only PC on a stack	ISR calls save the PC and PSW on a stack

ISR vs. subprogram/subroutine

The processor always executes subroutine calls as programmed.

However, the processor executes ISR only if triggered and the Interrupt Enable bit in the PSW is set.

Reasons for exception handling

- External reasons (asynchronous events): incoming data, device ready, mouse movement, …
- Internal reasons (synchronous events): system calls, debugging, change of privilege, …

External reasons for exceptions

RESET

- Reset of the processor, e.g., triggered by a button, power supply, watch dog timer, …

HALT

- Stop the execution of the processor, e.g., to avoid access conflicts on the system bus during DMA (direct memory access)

ERROR

- Call of an error handler routine, e.g., due to bus errors

Interrupt

- Interrupt request triggered by an external device, e.g., to announce incoming data of an input device
- 2 types: maskable/non maskable (NMI)

Internal reasons for exceptions

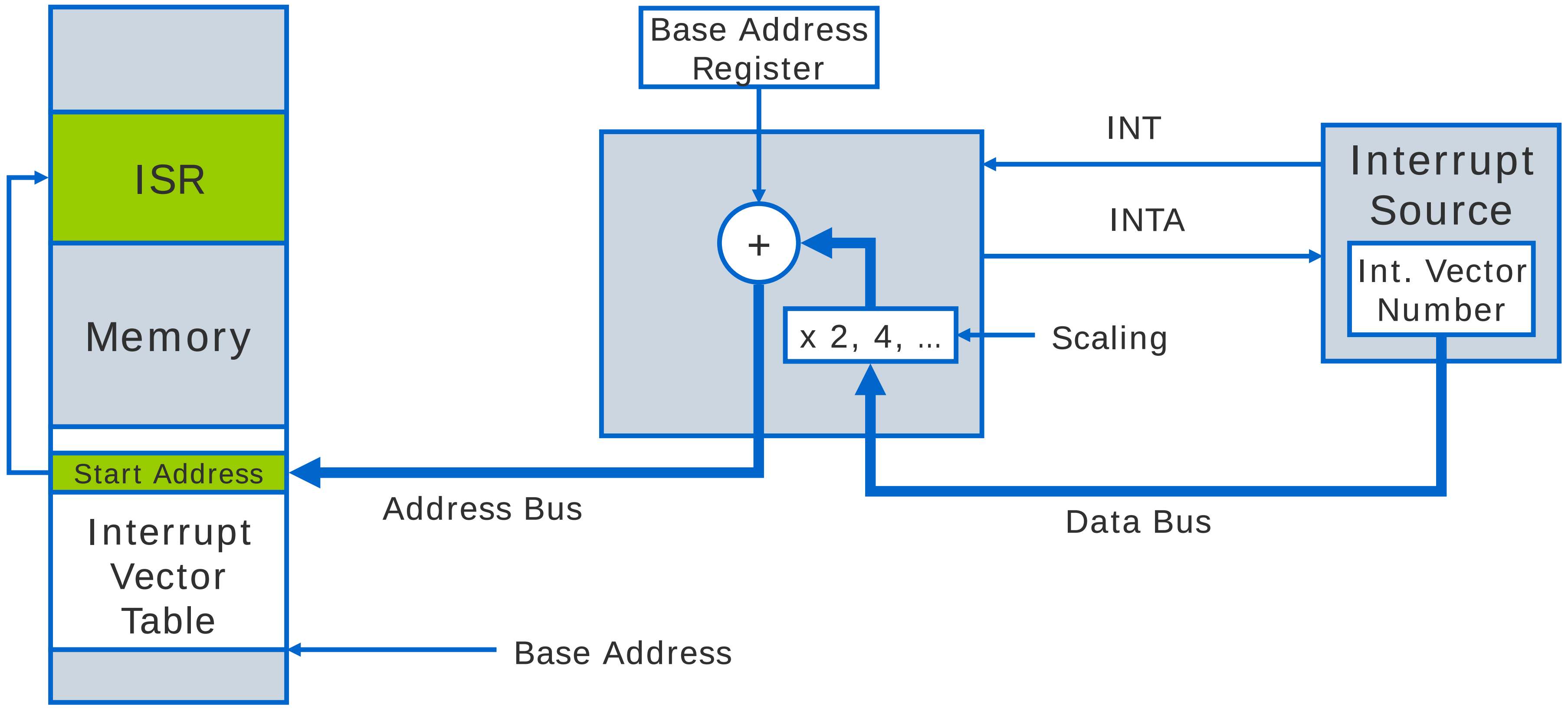
Software Interrupts

- `INT` instruction in the program triggers an interrupt (system calls, debugging, …)

Traps

- Exceptions caused by internal events, e.g., overflow, division by zero, stack overflow, …

Example: Calculation of the start address of an Interrupt Service Routine (ISR)



Typical steps of an ISR I

1. Interrupt activation
2. Finalize the instruction currently in execution
3. Check, if software interrupt or internal/external hardware interrupt
4. Check if Interrupt Enable bit is set
   ➔ allow interrupt
5. If it is a hardware interrupt: find source of interrupt, activate INTA (interrupt acknowledge)
6. Save PSW and PC on stack
7. Reset Interrupt Enable bit to avoid an additional interrupt in this stage

Typical steps of an ISR II

8. Calculate start address of ISR (e.g. based on the interrupt vector table) and load it into the PC
9. Execute the Interrupt Service Routine:
    - Push the used register on stack
    - Set the Interrupt Enable bit to allow other interrupts (i.e. interrupts can interrupt interrupts!)
    - Do the real work of the ISR
    - Pop the registers from stack
    - Return from interrupt handling using the IRET instruction
10. Restore PSW and PC and continue with the interrupted program

Be aware: if the ISR is too large it blocks the computer!

Interrupt vector table

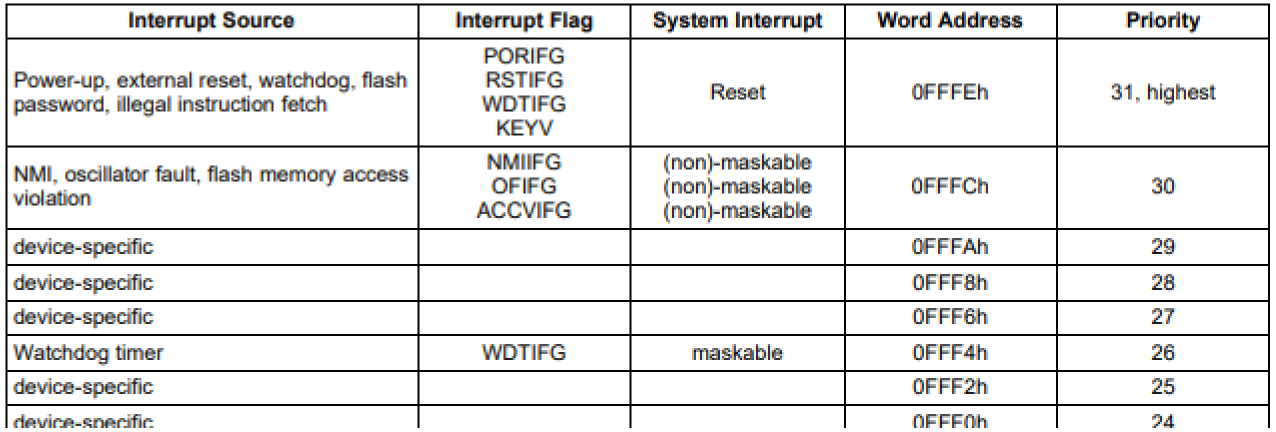
Typically, located at a well-known address, e.g., in ROM (starting at address 0000:0000 for 80X86 processors)

Contains the start addresses of the ISRs

The source of an interrupt creates an interrupt number pointing at the entry in the interrupt vector table

Can be way more complex …

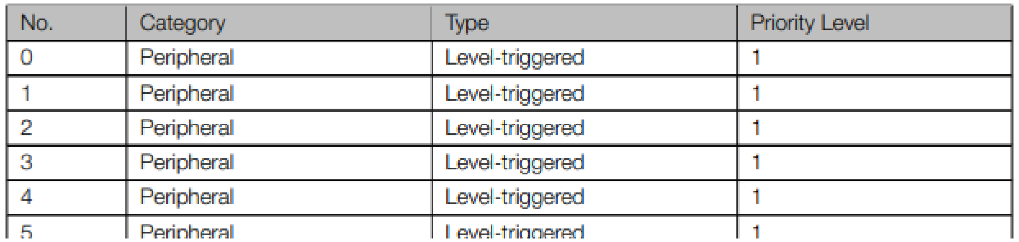
Examples of interrupt vector tables



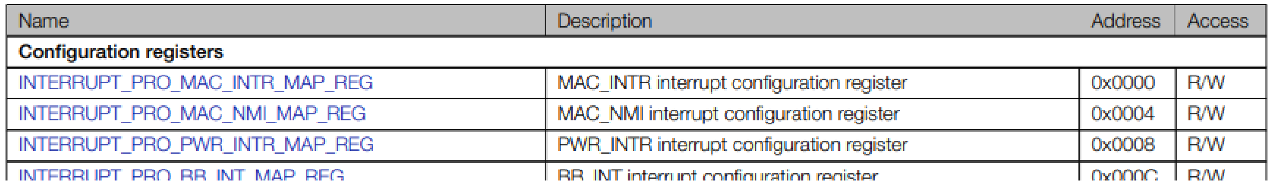
Interrupt Source	Interrupt Flag	System Interrupt	Word Address	Priority
Power-up, external reset, watchdog, flash password, illegal instruction fetch	PORIFG RSTIFG WDTIFG KEYV	Reset	0FFFEh	31, highest
NMI, oscillator fault, flash memory access violation	NMIIFG OFIFG ACCVIFG	(non)-maskable (non)-maskable (non)-maskable	0FFFCh	30
device-specific			0FFFAh	29
device-specific			0FFF8h	28
device-specific			0FFF6h	27
Watchdog timer	WDTIFG	maskable	0FFF4h	26
device-specific			0FFF2h	25
device-specific			0FFF0h	24

TI MSP430

https://www.ti.com/



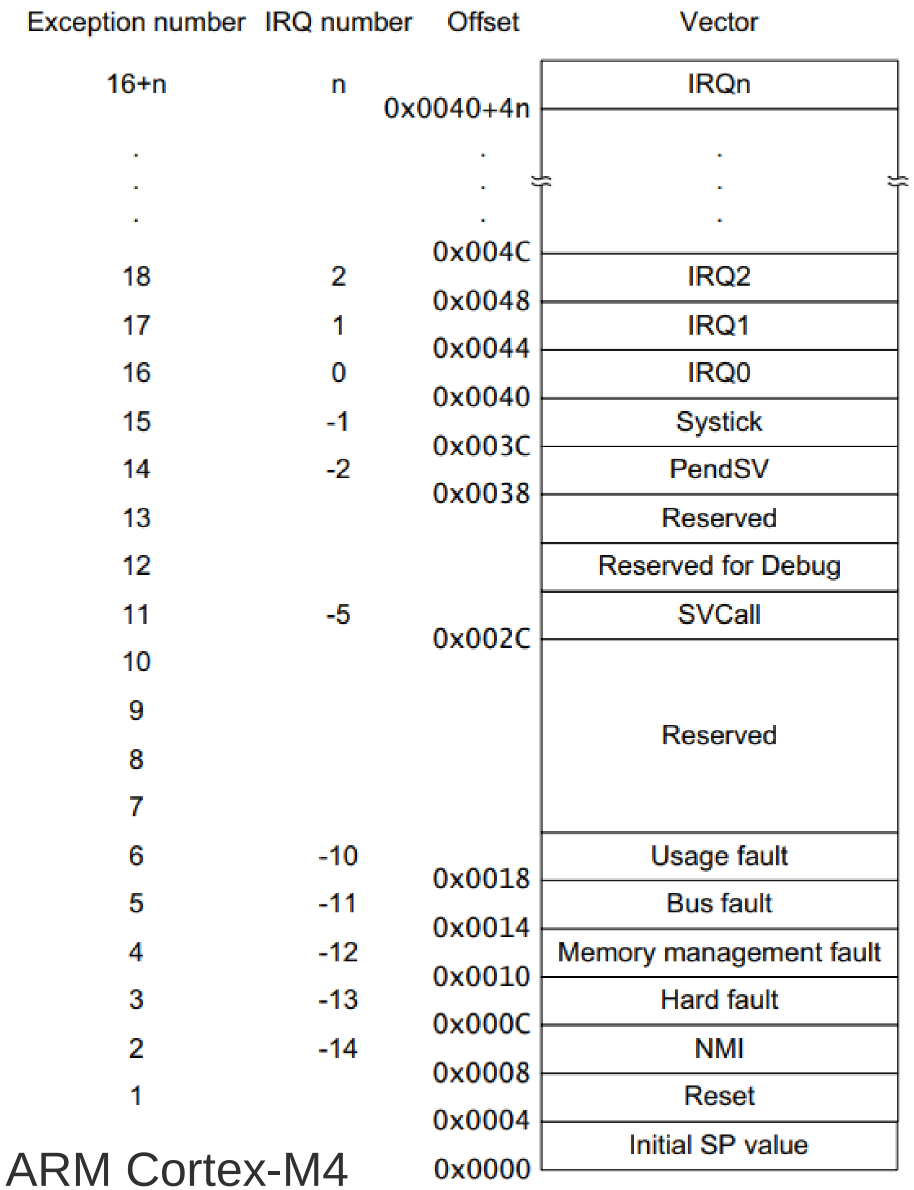
No.	Category	Type	Priority Level
0	Peripheral	Level-triggered	1
1	Peripheral	Level-triggered	1
2	Peripheral	Level-triggered	1
3	Peripheral	Level-triggered	1
4	Peripheral	Level-triggered	1
5	Peripheral	Level-triggered	1



Name	Description	Address	Access
Configuration registers			
INTERRUPT_PRO_MAC_INTR_MAP_REG	MAC_INTR interrupt configuration register	0x0000	R/W
INTERRUPT_PRO_MAC_NMI_MAP_REG	MAC_NMI interrupt configuration register	0x0004	R/W
INTERRUPT_PRO_PWR_INTR_MAP_REG	PWR_INTR interrupt configuration register	0x0008	R/W
INTERRUPT_PRO_BB_INT_MAP_REG	BB_INT interrupt configuration register	0x000C	R/W

Espressif ESP32-S2

https://www.espressif.com/



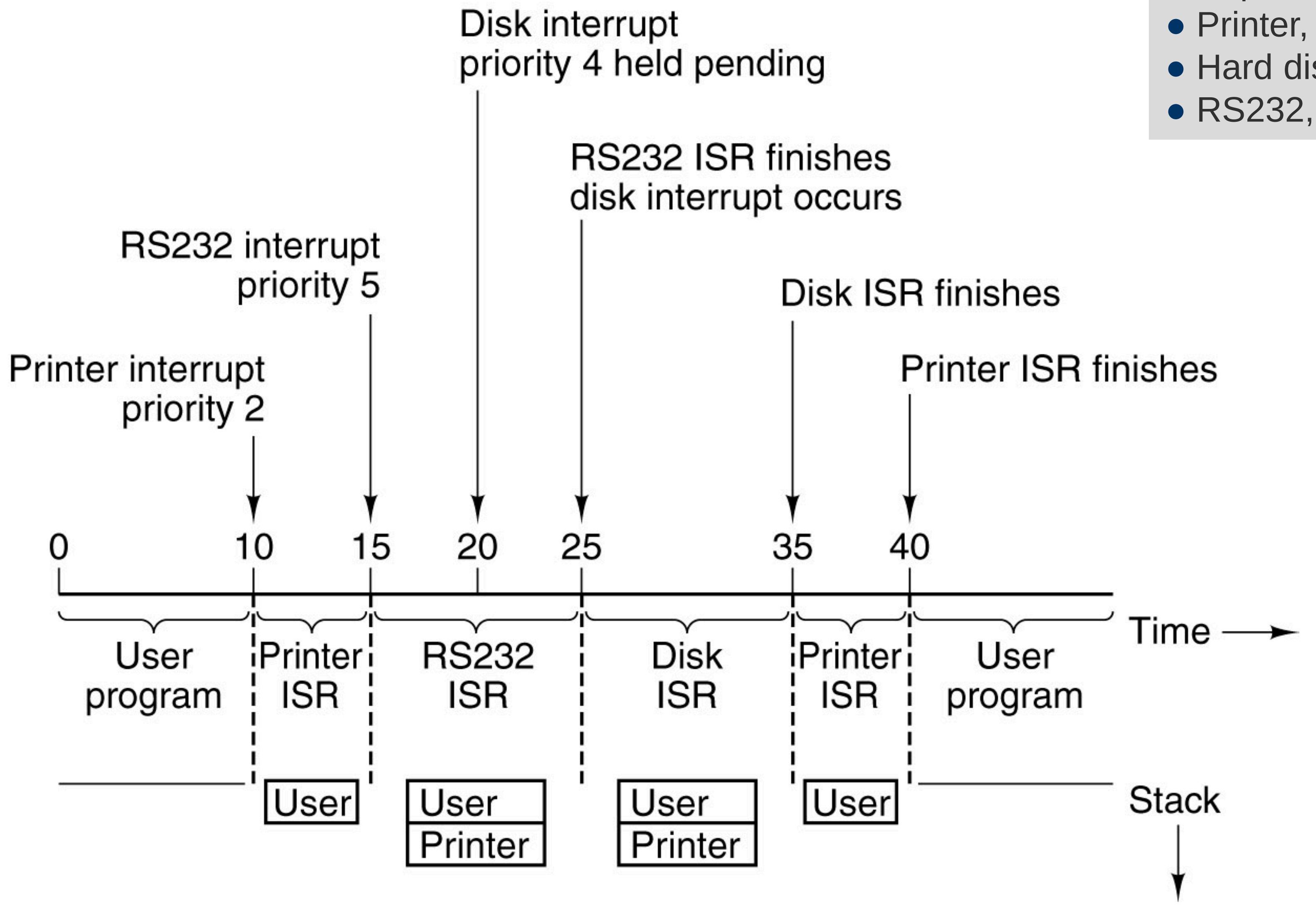
Exception number	IRQ number	Offset	Vector
16+n	n	0x0040+4n	IRQn
. . .		. . . 0x004C	. . .
18	2	0x0048	IRQ2
17	1	0x0044	IRQ1
16	0	0x0040	IRQ0
15	-1	0x003C	Systick
14	-2	0x0038	PendSV
13			Reserved
12			Reserved for Debug
11	-5	0x002C	SVCall
10			
9			Reserved
8			
7			
6	-10	0x0018	Usage fault
5	-11	0x0014	Bus fault
4	-12	0x0010	Memory management fault
3	-13	0x000C	Hard fault
2	-14	0x0008	NMI
1		0x0004	Reset
		0x0000	Initial SP value

ARM Cortex-M4

https://www.arm.com/

Time sequence of multiple interrupts

Computer with 3 I/O devices
- Printer, priority 2
- Hard disc, priority 4
- RS232, priority 5



Handling of multiple interrupt sources

Cyclic polling of interrupt sources by the interrupt controller (interrupt flag for each source in a status register of the controller).

If the interrupt flag for a component is set

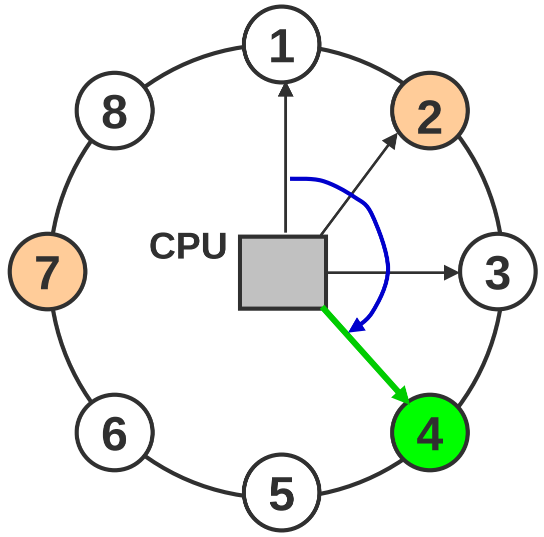
➡ Stop cyclic polling and start ISR for the source.

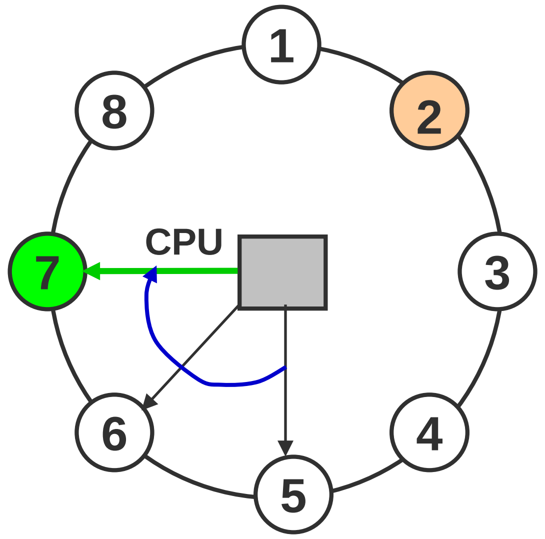
Several new/additional interrupt requests possible during or after the execution of an ISR.

➡ Two alternative ways of treating new/additional interrupts

Polling: 1. method

Continue cycling polling at the interrupt source following the last served source ➔ all interrupt sources have an equal chance of being served („fair“ processor sharing)



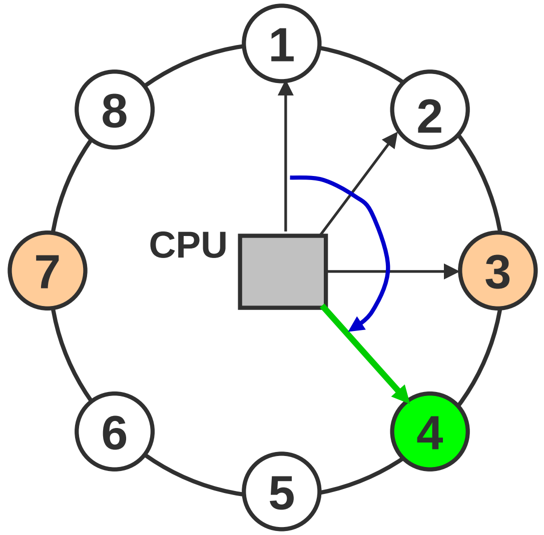


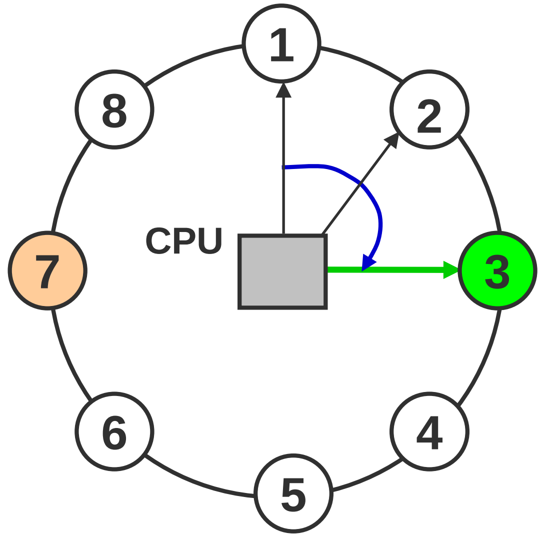
🟢 Interrupt source currently served

🟠 New interrupt requests during serving interrupt source 4

Polling: 2. method

Cyclic polling always starts at a pre-determined first source ➔ Different sources automatically get different priorities. Polling favors components with higher priority.





🟢 Interrupt source currently served

🟠 New interrupt requests during serving interrupt source 4

Polling

Priorities of the old 80286 (and many other x86):

Priority		Exception
0	RESET	Reset/initialization
1	TRAP	Exception during instruction execution
	INT	Software interrupt
2	TRACE	Single step execution
3	NMI	Non-maskable interrupt
4	...	Co-processor error
5	IRQ	Maskable interrupts

Disadvantage of polling:

Using software for cyclic prioritization and identification of interrupts is too time consuming.

Daisy chaining using hardware

Use specialized hardware for prioritization and identification of interrupts.

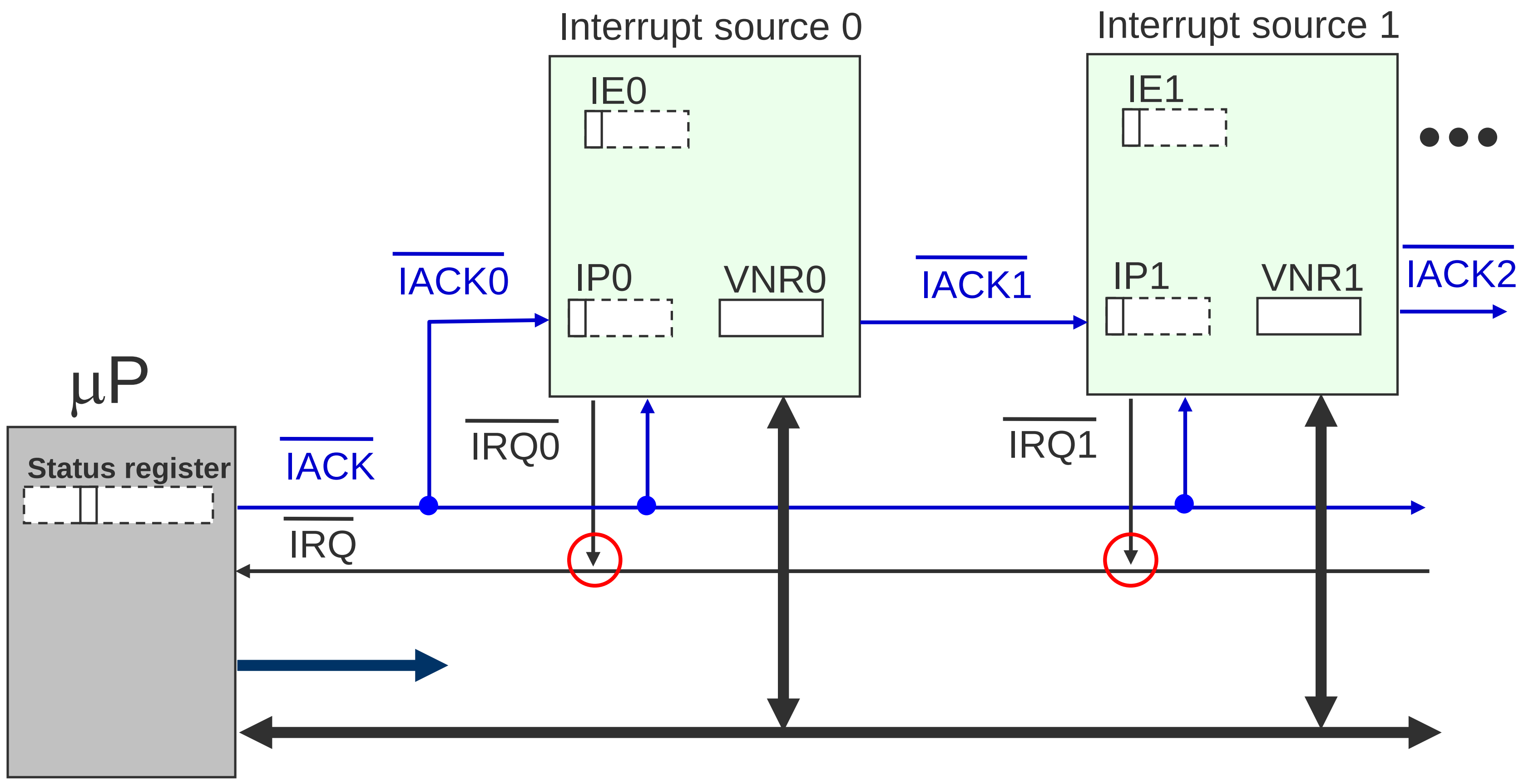
Example: Chaining of interrupt sources to a priority chain (Interrupt Daisy Chain).

Each source for an interrupt uses dedicated hardware for connecting with a successor and predecessor (decentralized prioritization)

The first source in the chain has automatically the highest priority.

The priority of the other sources depend on the position in the chain.

Daisy chaining using hardware

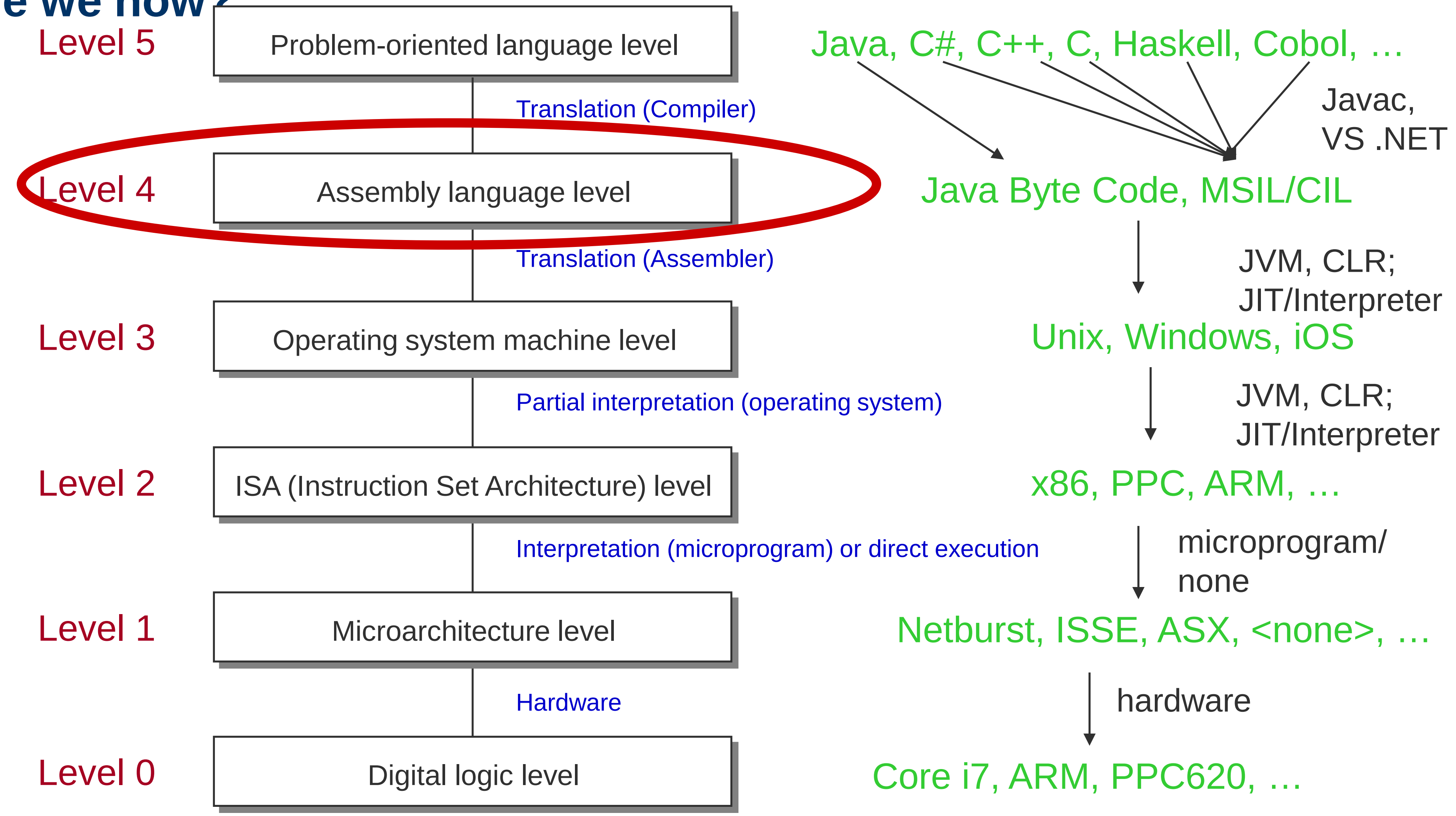


Questions & Tasks

- Name the key differences between a subprogram and an ISR!
- What is the purpose of an ISR?
- Could a computer operate without ISRs?
- Looking at the typical steps of an ISR – which step(s) should be uninterruptable?
- How to handle multiple interrupt sources?

ASSEMBLER

Where are we now?


Level 5
Problem-oriented language level
Translation (Compiler)
Level 4
Assembly language level
Translation (Assembler)
Level 3
Operating system machine level
Partial interpretation (operating system)
Level 2
ISA (Instruction Set Architecture) level
Interpretation (microprogram) or direct execution
Level 1
Microarchitecture level
Hardware
Level 0
Digital logic level
Java, C#, C++, C, Haskell, Cobol, …
Javac, VS .NET
Java Byte Code, MSIL/CIL
JVM, CLR; JIT/Interpreter
Unix, Windows, iOS
JVM, CLR; JIT/Interpreter
x86, PPC, ARM, …
microprogram/ none
Netburst, ISSE, ASX, <none>, …
hardware
Core i7, ARM, PPC620, …

Compiler vs. Assembler

Assembler

- Source: symbolic representation of a machine language (assembly language)
- Destination: numerical representation of the machine language (instructions from ISA)
- Examples: inline assembler in Visual Studio, MASM, ilasm, asm (gcc, Linux), MMIXal, nasm, ...

Compiler

- Source: high-level language (depends on the definition of „high“ ...), e.g., C, Java, C#, Cobol, Modula, C++, ...
- Destination: assembler language or (built-in assembler) numerical representation of the machine language (instructions from ISA)
- Examples: C#-Compiler in Visual Studio, gcc, cc, javac, ...

Assembler language

- Pure assembler language: 1:1 mapping onto ISA instructions
- But additionally: symbolic names, addresses, labels

Reasons for an assembler level

Full access to HW features

- (almost) all (visible) registers are exposed to the assembler language, all flags can be read or set, many „hidden“ features can be used
  - E.g., try accessing the Pentium performance counters from within Java

Performance

- Optimized code for special purposes
  - Real-time: exact number of CPU cycles can be counted, guaranteed access times to registers, deterministic response times of sub-routines (again: try Java and real-time – and see what a garbage collector does...)
  - Low memory footprint: no useless overhead, optimized loops, etc.

But much harder to program in assembler ....

- Thus typically combined with, e.g., C – only performance critical parts of a program will be tuned via (inline) assembler (https://en.wikipedia.org/wiki/Inline_assembler)

Examples for assembler: N = I + J

Pentium

```
FORMULA:
    MOV   EAX,I     ; register EAX = I
    ADD   EAX,J     ; register EAX = I + J
    MOV   N,EAX     ; N = I + J

I   DW   3          ; reserve 4 byte initialized to 3
J          DW      4         ; reserve 4 byte initialized to 4
N          DW      0         ; reserve 4 byte initialized to 0
```

SPARC

```
FORMULA:
    SETHI %HI(I),%R1                 ! R1 = high-order bits of the address of I
    LD    [%R1+%LO(I)],%R1  ! R1 = I
    SETHI %HI(J),%R2                 ! R2 = high-order bits of the address of J
    LD    [%R2+%LO(J)],%R2  ! R2 = J
    NOP                     ! wait for J to arrive from memory
    ADD   %R1,%R2,%R2                ! R2 = R1 + R2
    SETHI %HI(N),%R1                 ! R1 = high-order bits of the address of N
    ST    %R2,[%R1+%LO(N)]  ! N = I + J

I:   .WORD 3                         ! reserve 4 byte initialized to 3
J:   .WORD 4                         ! reserve 4 byte initialized to 4
N:   .WORD 0                         ! reserve 4 byte initialized to 0
```

Pseudo instructions

Pseudo instructions or assembler directives

- Help a lot for assembler programming
- Depend on designer of the assembler, not the ISA

Examples (MASM for Pentium)

- DB	allocate storage for one or more (initialized) bytes
- DW	allocate storage for one or more (initialized) 32 bit words
- PROC	start a procedure
- MACRO	start a macro definition
- INCLUDE	fetch and include another file
- IF	start conditional assembly based on a given expression
- PUBLIC	export a name defined in the module

Macros vs. procedures

Example macro: swap P, Q

```
SWAP      MACRO
            MOV EAX,P
            MOV EBX,Q
            MOV Q,EAX
            MOV P,EBX
          ENDM
```

Differences	Macro	Procedure
When is the call made?	During assembly	During execution
Is the body inserted into the object program every place the call is made?	Yes	No
Is a call instruction inserted into the object program and later executed?	No	Yes
Must a return instruction be used after the call is done?	No	Yes
How many copies of the body appear in the object program?	One per macro call	1

Macros are „textually“ inserted into the assembler program each time a call is made, formal parameters are converted into actual parameters (i.e., the above macro can be used for SWAP A,B as well as SWAP X,Y)

The assembler process

Step-by-step translation does not work

- Forward references: symbols used before being defined ...

Solution: two-pass translator or single-pass plus conversion into intermediate format

- Pass: reading the source

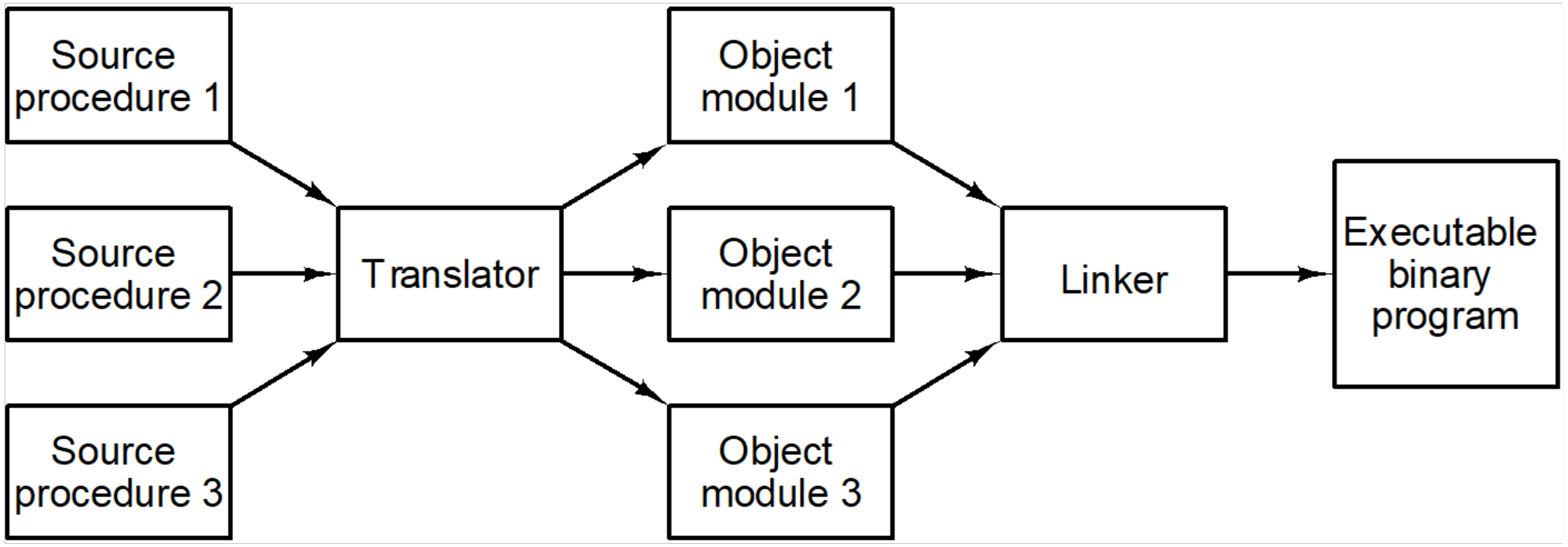
First step

- Check syntax
- Create a symbol table, opcode table, literal table
- Check instruction length (opcode, operands, ...)

Second step

- Generate object code (*.o, *.obj, ...)
- Generate information for linker

Generation of an executable binary program

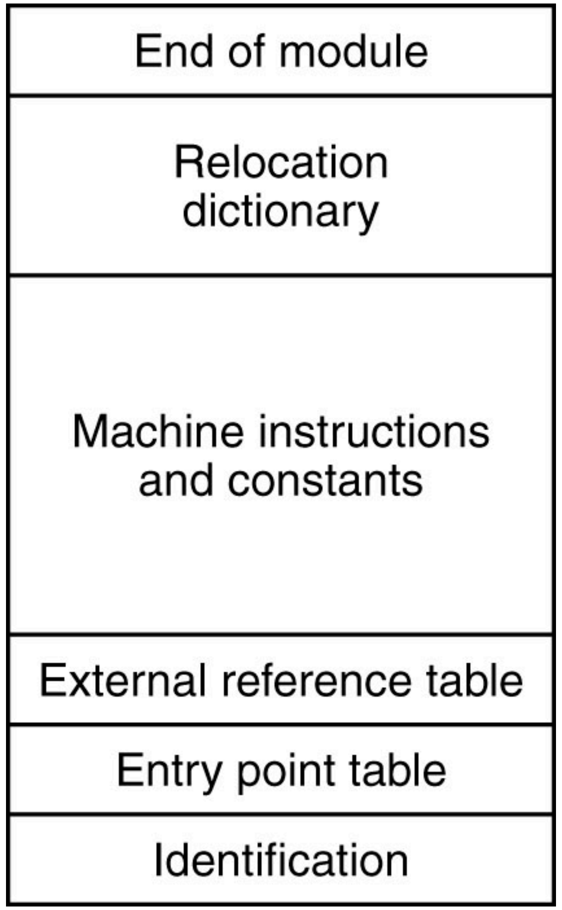


Example Structure of an Object Module / File

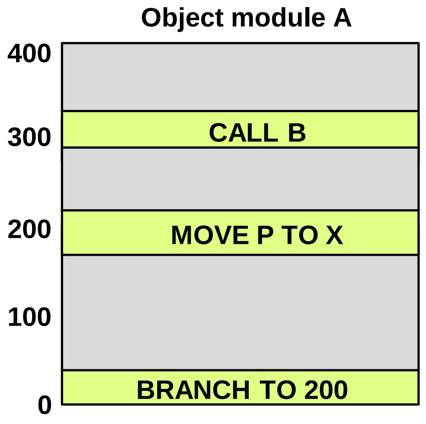
Different formats exist (e.g. COFF, ELF)

Relocation often done by MMU (or code is position independent) – but also loader can relocate

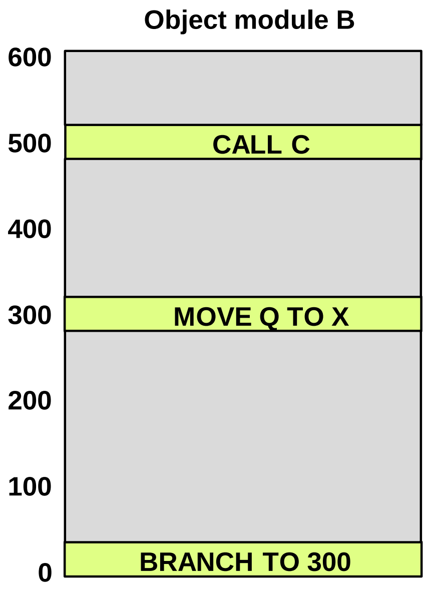
https://en.wikipedia.org/wiki/Object_file

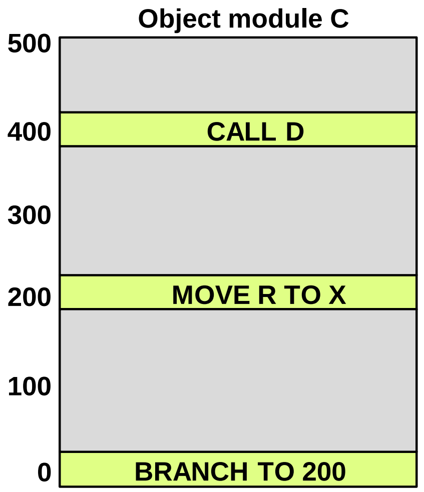


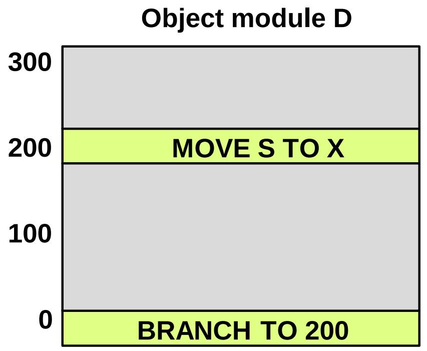
Example object modules



Each module has its own address space starting at 0

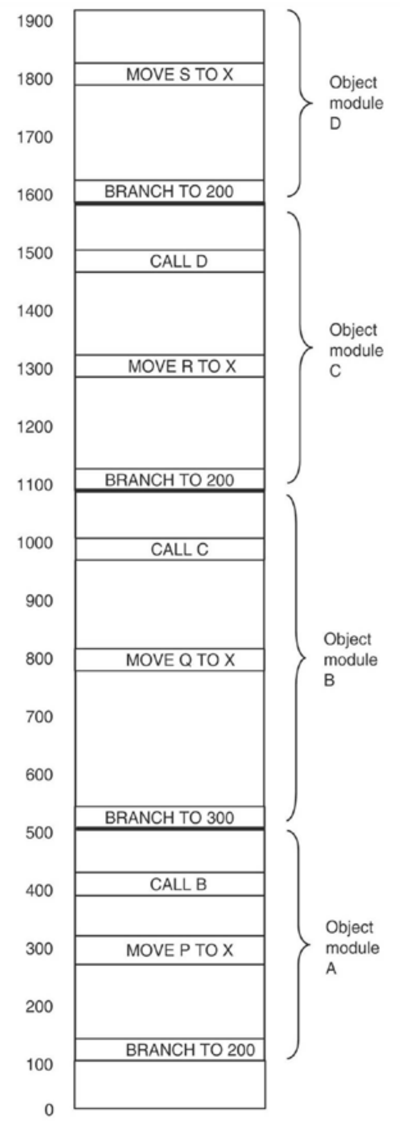


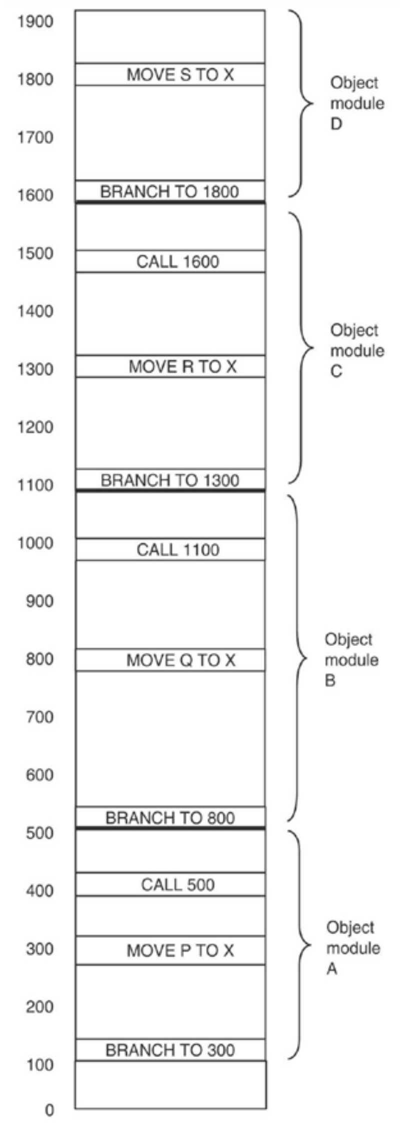




Objects before/after linking and relocation

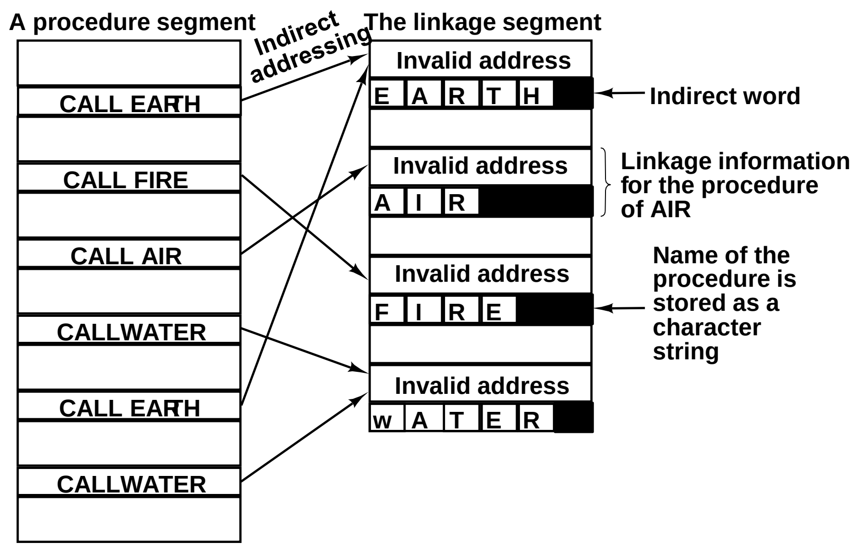
Before relocation and linking



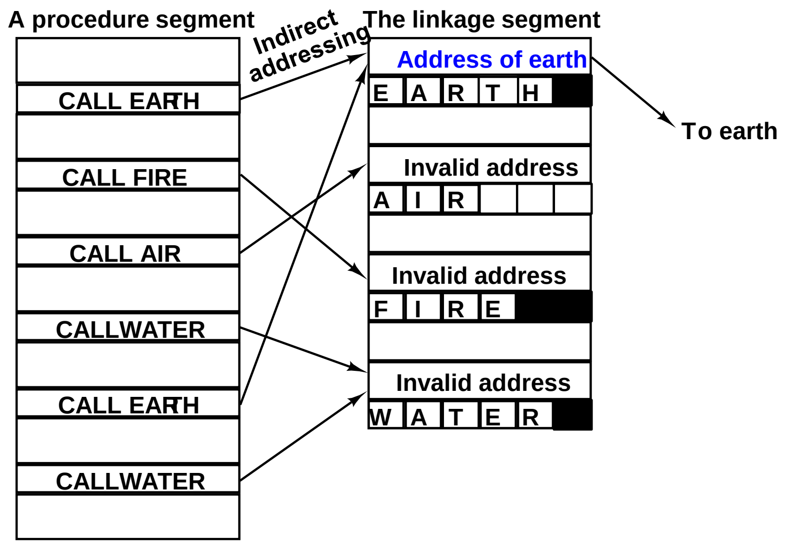


After relocation and linking

Dynamic linking



Before EARTH is called

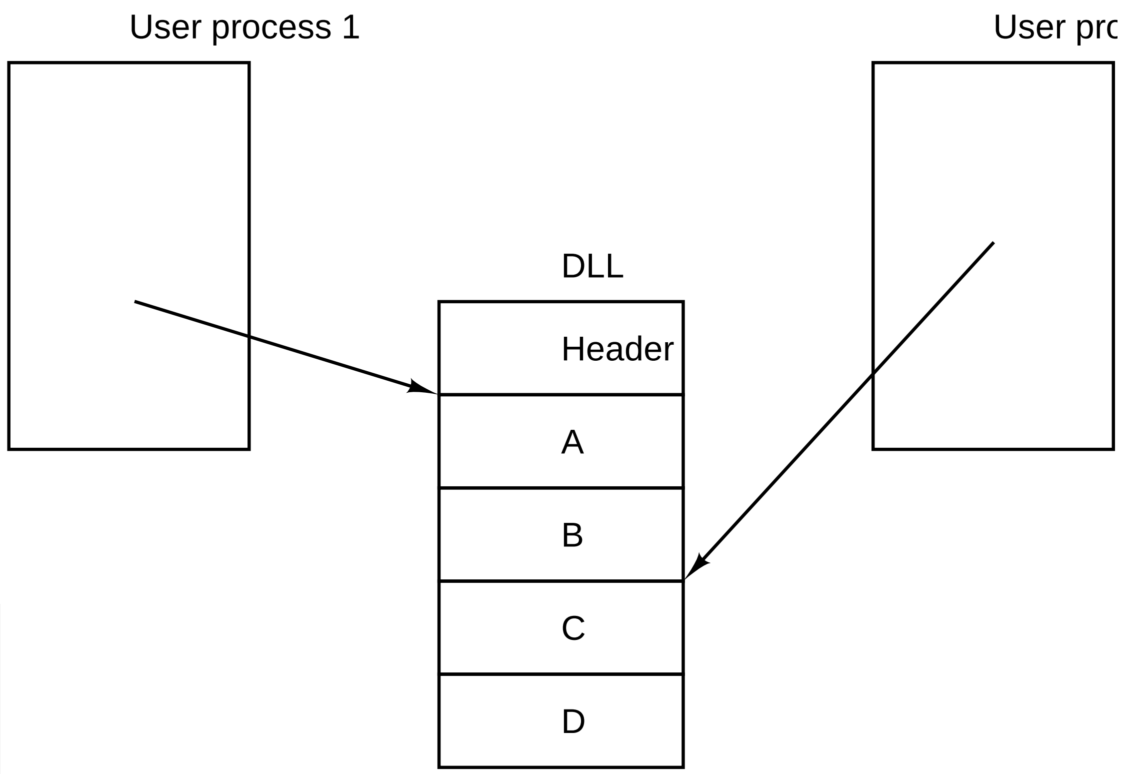


After EARTH has been called and linked

Shared libraries

DLL (Dynamic Link Library, Windows), shared library (Unix)

- Save a lot of memory as they appear only once
- Many processes „share“ the same code (instructions)



A lot more info given in OS and compiler courses!

Questions & Tasks

- Why using assembler today?
- Why using macros? Typically they require more space…
- When is the starting address of an object determined? (Many answers possible … When do we really need the address at the latest?)

Summary

Soft-/Hardware boundary

Complex Instruction Set Computer (CISC)

Reduced Instruction Set Computer (RISC)

Examples of ISA

Instructions formats

Addressing formats

Types of instructions

Procedures, Traps, Interrupts & Co.

Assembler