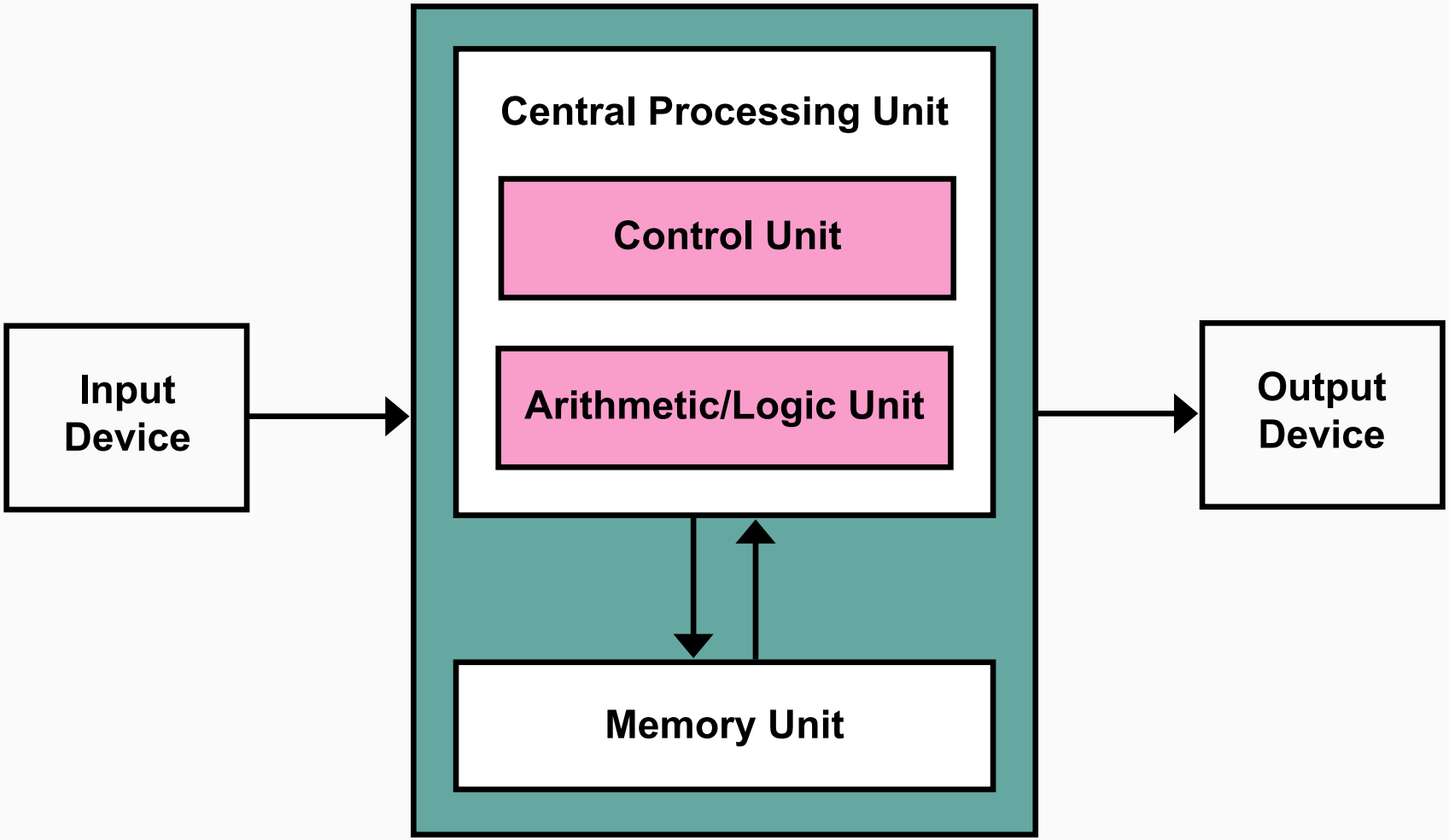
INF108: Compilation

Louis Jachiet

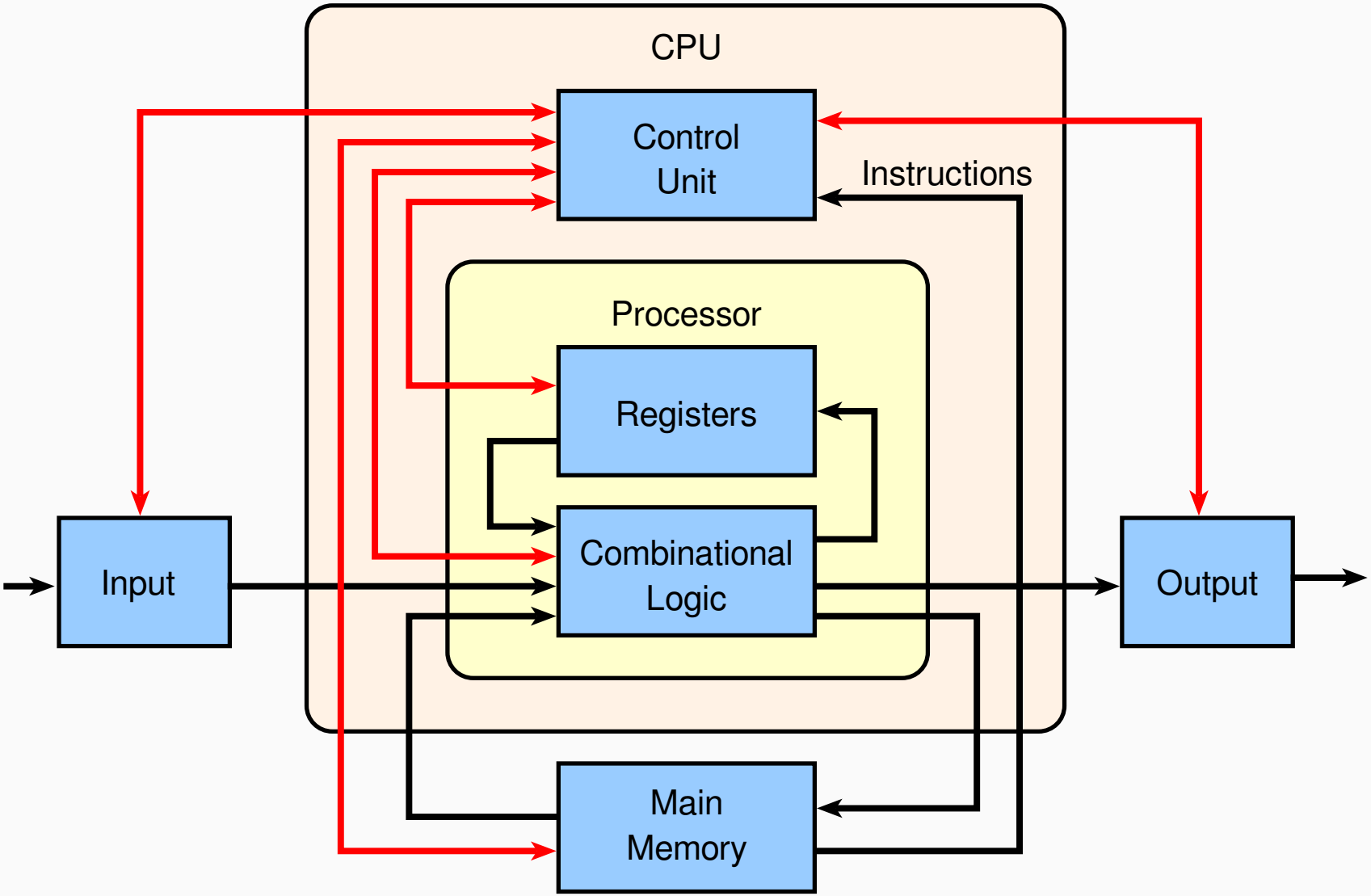
General architecture of computers

The von Neumann architecture



Von Neumann Architecture on Wikipedia

The von Neumann architecture



Computer Architecture on Wikipedia

A modern CPU

Introducing 11th Gen Intel® Core™ Processor

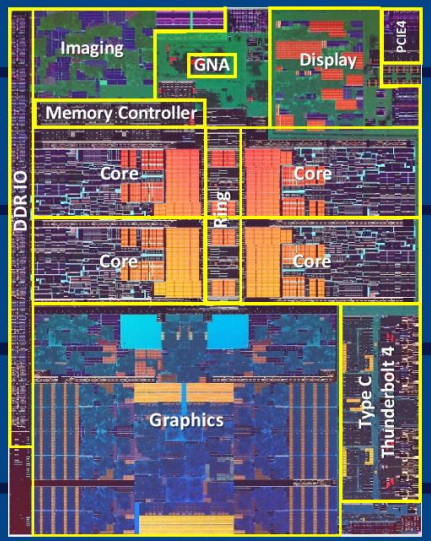
New Willow Cove Cores
Up to 4 Cores / 8 Threads
Up to 4.8GHz

New Converged Chassis Fabric
High Bandwidth / Low Latency
IP and Core Scalable

New Memory Controller
LP4/x-4266 4x32b up to 32GB
DDR4-3200 2x64b up to 64GB

1st Integrated Thunderbolt™ 4
Full 4x DP/USB/PCIe mux on-die
Up to 40Gbps bi-directional per port

1st Integrated PCIe Gen 4 (CPU)
Low Latency, High Bandwidth
SSD or Discrete Graphics Direct CPU Attach



New Iris® Xe Graphics
Up to 96EU – Up to 2x Higher Performance
Intel® Deep learning Boost: DP4A for AI

New 2x MEDIA Encoders
Up to 4K60 10b 4:4:4
Up to 8K30 10b 4:2:0

New 4 x Display Pipes
Up to 1 x 8K60 or 4 x 4K60
DP1.4 HBR3, BT.2020

New Image Processing Unit (IPU6)
Video up to 4K90 resolutions (initially 4K30)
Still image up to 42 megapixels (initially 27MP)

New GNA 2.0
Enhanced Power Management
Autonomous DVFS

For more complete information about performance and benchmark results, visit www.intel.com/11thgen (configuration details in section 3).

intel

Instruction Set Architecture

Instruction Set Architecture (ISA)

ISA is an abstract model of a CPU specifying how it operates.

Organization of a CPU

A (simplified) ISA comprises:

- an access to the general memory
- registers (a local memory)
- a machine language

Machine language

In a machine language, programs are sequence of simple instruction, where each instruction is composed an *opcode* (what to do?) and *operands* (what are the arguments).

A very general ISA

In general, we have:

- some general purpose registers R0 ...
- a special PC register (current instruction)
- some instructions that modify registers
- some instructions to get/set memory in RAM
- some instructions to branch
- some specialized instructions (hardware devices, coprocessor, or other)

RISC vs CISC!

Reduced Instruction Set Computer (RISC)

- easier to learn and to optimize via algorithms
- faster per instruction
- smaller chips

Notable examples: MIPS & ARM

Complex Instruction Set Computer (CISC)

- more computation per instruction
- smaller code
- microprogramming of the CPU
- well optimized (advanced) computations

Notable examples: x86-64

General architecture of computers

MIPS

MIPS

- MIPS is a RISC (very simple ISA)
- exists in several versions $\Rightarrow$ we will use R2000
- 32 "general purpose" registers of 32 bits ($2^{32} = 4\,294\,967\,296$)
- threes special registers: `$pc`, `$hi`, `$lo`
- instructions are 32 bits (i.e. 4 bytes)

Register (R) Instructions

$\texttt{\$rd} = \mathit{func}(\mathit{shift}(\texttt{shamt}, \texttt{\$rs}), \texttt{\$rt})$

opcode (6)	**rs** (5)	**rt** (5)	**rd** (5)	**shamt** (5)	**func** (6)
0	modified	arg1	arg2	shift	operation

Example:

$0x00c23021 = 0b000000\ 00110\ 00010\ 00110\ 00000\ 100001$ is

$\texttt{\$r6} =$ unsigned add($\texttt{\$r6}$,$\texttt{\$r2}$)

Immediate (I) Instructions

$\texttt{\$rt} = opcode(\texttt{\$rs}, IMM)$

opcode (6)	**rs** (5)	**rt** (5)	**IMM** (16)
function	modified	arg1	value

Example:

$0x27a50004 = 0b001001\ 11101\ 00101\ 0000000000000100$ is

$\texttt{\$r5}$ = unsigned add $\texttt{\$r29} + 4$

Jump (J) Instructions

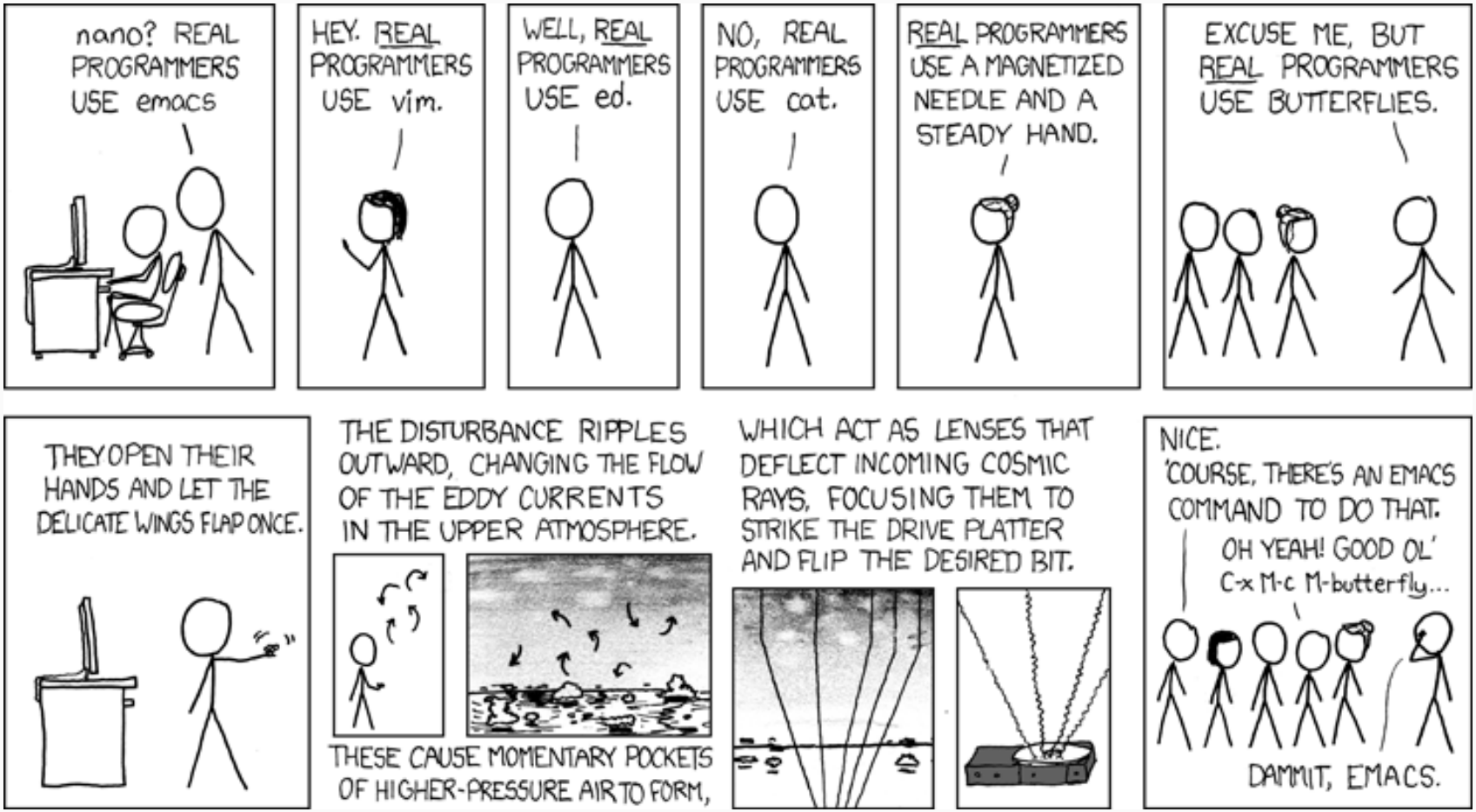
$\$pc = (\$pc \& 0xf0000000) + addr * 4$

opcode (6)	**addr** (26)
type	addr

Example: $0x0c100009 = 0b\,110000\,01000000000000000001001$ is

$\$pc$ = jump to $0x400024$ relative $\$pc$

Unrelated



XKCD 378

No one really writes code in binary...

Binary is:

- hard to write
- hard to read
- relatives addresses needs to be recomputed for every instruction change
- modularity is almost impossible

MIPS assembler!

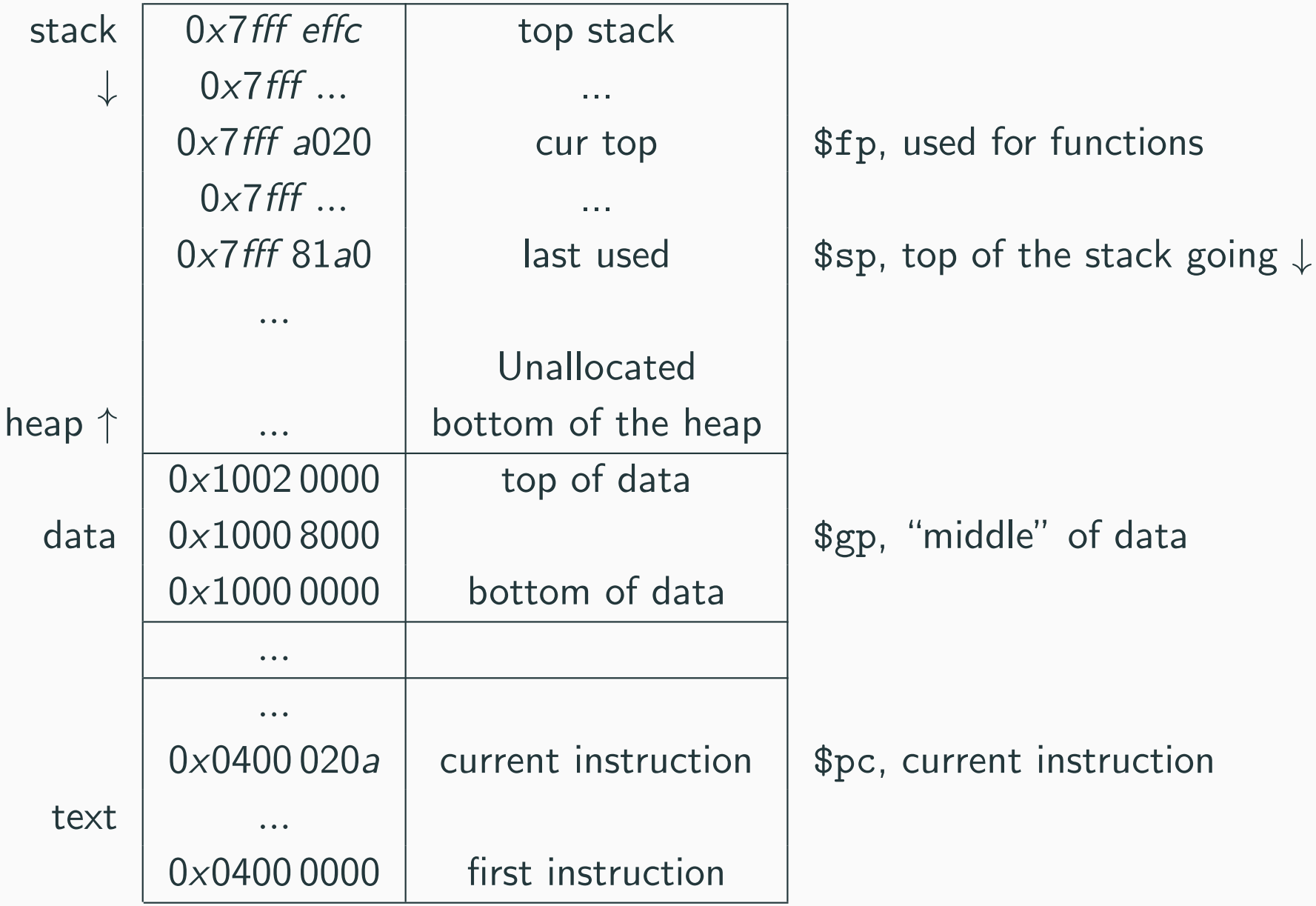
```
.text
main:
    ori $t1, $zero, 42 # t1 = 42
loop:
    ori $v0, $zero, 1          # system call 1 (print_int)
    addu $a0, $zero, $t1        # and print the value a0 = r0
    syscall                    #
    ori $v0, $zero, 11         # system call 11 (print_char)
    ori $a0, $zero, 10         # and print the char 10 = \n
    syscall                    #
    add $t1, $t1, -1           # t1--
    bgez $t1 loop              # if t1>=0 restart
end:
    li $v0, 10                 # system call code for exit
    syscall                    # exiting
```

MIPS memory

0	zero	always 0
1	at	used by pseudo inst.
2-3	v0-v1	results of fonctions
4-7	a0-a3	arguments of fonctions
8-15	t0-t7	temporaries
16-23	s0-s7	saved
24-25	t8-t9	temporaries
26-27	k0-k1	kernel
28	gp	global pointer
29	sp	stack pointer
30	fp	frame pointer
31	ra	return address

Also: `$hi`, `$lo`, `$pc`

MIPS memory


stack
↓
heap ↑
data
text
0x7fff effc
top stack
0x7fff ...
...
0x7fff a020
cur top
$fp, used for functions
0x7fff ...
...
0x7fff 81a0
last used
$sp, top of the stack going ↓
...
Unallocated
...
bottom of the heap
0x1002 0000
top of data
0x1000 8000
$gp, "middle" of data
0x1000 0000
bottom of data
...
...
0x0400 020a
current instruction
$pc, current instruction
...
0x0400 0000
first instruction

MIPS logic & arithmetic

Standard manipulation

- arithmetic: ADD / SUB / DIV / MUL
- unsigned version +U, float version +.s
- logic: AND / OR / XOR /
- shift: SRL / SLL / SRA / SLA
- comparison: SLT & variants
- set values: LUI / ORI / LA
- to/from RAM: SW / LW / SB / LB
- unconditional jump: J / JAL / JR
- conditional jump: BEQ / BGEZ / BGTZ / BLEZ / ...

SPIM syscalls

1	print int
2	print float
3	print double
4	print string
5	read int
6	read float
7	read double
8	read string
9	sbrk
10	exit
11	print char
12	read char

Making a syscall

Syscall type in `$v0`, arg in `$a0`
result in `$v0` and then `syscall` instruction

MIPS data

Useful assembly directives:

- `.data` goes to the data segment
- `.text` goes to the code segment
- `myLabel:` mark a position
- `.asciiz`, `.ascii` strings
- `.word`, `.byte`, `.half` data of a given size
- `.align n`
- `.globl` symbol accessible from other files

```
.data
myStr: .asciiz "my string\n "
.text
main:
    la $a0, myStr      # address of string into £a0
```

How to encode basic constructs?

Do-While loops

```
loop:
    #do stuff
    bgtz $t1 loop # continue when t1>0
    #after the loop
```

While loop

```
loop:
    blez $t1 endloop # continue when t1>0
    #do stuff
    j loop
endloop:
    #after the loop
```

How to encode basic constructs?

If-then-else

```
        #do test
        bgtz $t1 else #
then:
        # here we put what to do when then
        j endif
else:
        # here we put what to do when else
endif:
        # rest of the program
```

How to encode basic constructs?

Function

```
        # to call the function
        jal myFunction
        # jal set $ra to be the next instruction
        add $1, $0, $0 # <- this for instance

myFunction:
        # content of the function
        # return to $ra
        jr $ra
```

How to encode basic constructs?

Function

```
        # to call the function
        jal myFunction
        # jal set $ra to be the next instruction
        add $1, $0, $0 # <- this for instance

myFunction:
        # content of the function
        # return to $ra
        jr $ra
```

What happens for functions that call functions?

How to encode basic constructs?

Function

```
myFunction:
        addi $sp, $sp, -4
        sw $ra, 0($sp) #store ra on the stack
        # content of the function
        lw $ra, 0($sp) # retrieve ra from the stack
        addi $sp, $sp, 4
        jr $ra
```

How to encode basic constructs?

Function

```
myFunction:
        addi $sp, $sp, -4
        sw $ra, 0($sp) #store ra on the stack
        # content of the function
        lw $ra, 0($sp) # retrieve ra from the stack
        addi $sp, $sp, 4
        jr $ra
```

What happens for functions that modify registers or store data on the stack?

MIPS memory

0	zero	always 0
1	at	used by pseudo inst.
2-3	v0-v1	results of fonctions
4-7	a0-a3	arguments of fonctions
8-15	t0-t7	temporaries
16-23	s0-s7	saved
24-25	t8-t9	temporaries
26-27	k0-k1	kernel
28	gp	global pointer
29	sp	stack pointer
30	fp	frame pointer
31	ra	return address

Also: `$hi`, `$lo`, `$pc`

MIPS assembler!

```
.text
main:
    ori $t1, $zero, 42 # t1 = 42
loop:
    ori $v0, $zero, 1          # system call 1 (print_int)
    addu $a0, $zero, $t1        # and print the value a0 = r0
    syscall                    #
    ori $v0, $zero, 11         # system call 11 (print_char)
    ori $a0, $zero, 10         # and print the char 10 = \n
    syscall                    #
    add $t1, $t1, -1           # t1--
    bgez $t1 loop              # if t1>=0 restart
end:
    li $v0, 10                 # system call code for exit
    syscall                    # exiting
```

XSpim demo!

Should you program in assembly?

In real life: NO:

- harder to write
- harder to read
- faster code but compilers are good and improving the algorithm is usually a better idea

---

Here: **YES**

- learning how a CPU works
- understanding better the performance bottlenecks of high-level constructs
- useful for the compiler project :)