Computer Architectures

Arithmetic + Processor

Richard Šusta, Pavel Píša

Czech Technical University in Prague, Faculty of Electrical Engineering

Floating point arithmetic operations overview

Conversion: **to/from *integer, double, float***

- shift of mantissa according to exponent

Addition: $A \cdot z^a, B \cdot z^b, b < a$ unify exponents

$B \cdot z^b = (B \cdot z^{b-a}) \cdot z^{b-(b-a)}$ by shift of mantissa

$A \cdot z^a + B \cdot z^b = [A + (B \cdot z^{b-a})] \cdot z^a$ sum + normalization

Subtraction: unification of exponents, subtraction and normalization

Multiplication: $A \cdot z^a \cdot B \cdot z^b = A \cdot B \cdot z^{a+b}$

$A \cdot B$ - normalize if required

$A \cdot B \cdot z^{a+b} = A \cdot B \cdot z \cdot z^{a+b-1}$ by left shift

Division: $A \cdot z^a / B \cdot z^b = A/B \cdot z^{a-b}$

A/B - normalize if required

$A/B \cdot z^{a-b} = A/B \cdot z \cdot z^{a-b+1}$ by right shift

Example: a + b = 1000*pi + e/20

1.		**Float 6.5 digits**	**to int**	<8 388 608;16 777 216) = <2^23;2^24)
af	1000*pi	~3141.59<u>3</u>	* 2^12	**12867964**,928
bf	e/20	~0,135914<u>1</u>	* 2^26	**9121040**,8525824

2.	**Convert to binary**		but we should add
ax	**12867965**	1100 0100 0101 1001 0111 1101	. after 12.bit <- * 2^12
bx	**9121041**	1000 1011 0010 1101 0001 0001	. after 26. bit <- * 2^26

3.	**Binary number**	**exponent**
a	1100 0100 0101.1001 0111 1101	2^0
b	00.00 1000 1011 0010 1101 0001 0001	2^0

4.	**Normalized binary number**	**exponent**
a	1.100 0100 0101 1001 0111 1101	* 2^11, 12+11=23
b	1.000 1011 0010 1101 0001 0001	* 2^-3, 26-3 = 23

Checking Intermediate Results

1.100 0100 0101 1001 0111 1101 * 2^11

= (12867965 * 2^-23) * 2^11 *- actually stored value*

= **1,53398**09656143188476 6 * 2^11

= **3141,593**017578125000000768

(1000*pi=**3141,592**65358979323846264)

1.000 1011 0010 1101 0001 0001 * 2^-3

= (9121041 * 2^-23) * 2^-3

= **1.087312**81757354736328 * 2^-3

= **0.1359141**0219669342041

(e/20=0,**1359140**9142295226177)

Calculation performed by ***SpeedCrunch 0.12 - http://speedcrunch.org/***

Preparing for Addition

5.	Normalizované binární číslo	exp.
a	1.100 0100 0101 1001 0111 1101	* 2^11
+ b	1.000 1011 0010 1101 0001 0001	* 2^-3

6.	Na stejné exponenty	exp.
a	1.100 0100 0101 1001 0111 1101	* 2^11
+ b	0.000 0000 0000 0010 0010 1100 *10110100010001*	* 2^11

*For number **b**, the binary dot is shifted 14 points to the left, i.e., the difference of the exponents 11- (-3). Red-marked bits run out of the range -> loss of accuracy.*

7.	Součet	expt
a	1.100 0100 0101 1001 0111 1101	* 2^11
+ b	0.000 0000 0000 0010 0010 1100 ~~*10110100010001*~~	* 2^11
a+b	1.100 0100 0101 1011 1010 1001	* 2^11

Result and its double type check

a+b = 1.100 0100 0101 1011 1010 1001 * 2^11

= (12868521 * 2^-23) * 2^11

= **1.534047**24597930908203 * 2^11

= **3141,728**7597656249999744

Original numbers a and b added as double
=3141.593 + 0,1359141 = **3141,728**9141

= 1.100010001011011101010**1010** * 2^11 its real value

= 3141,72900390625

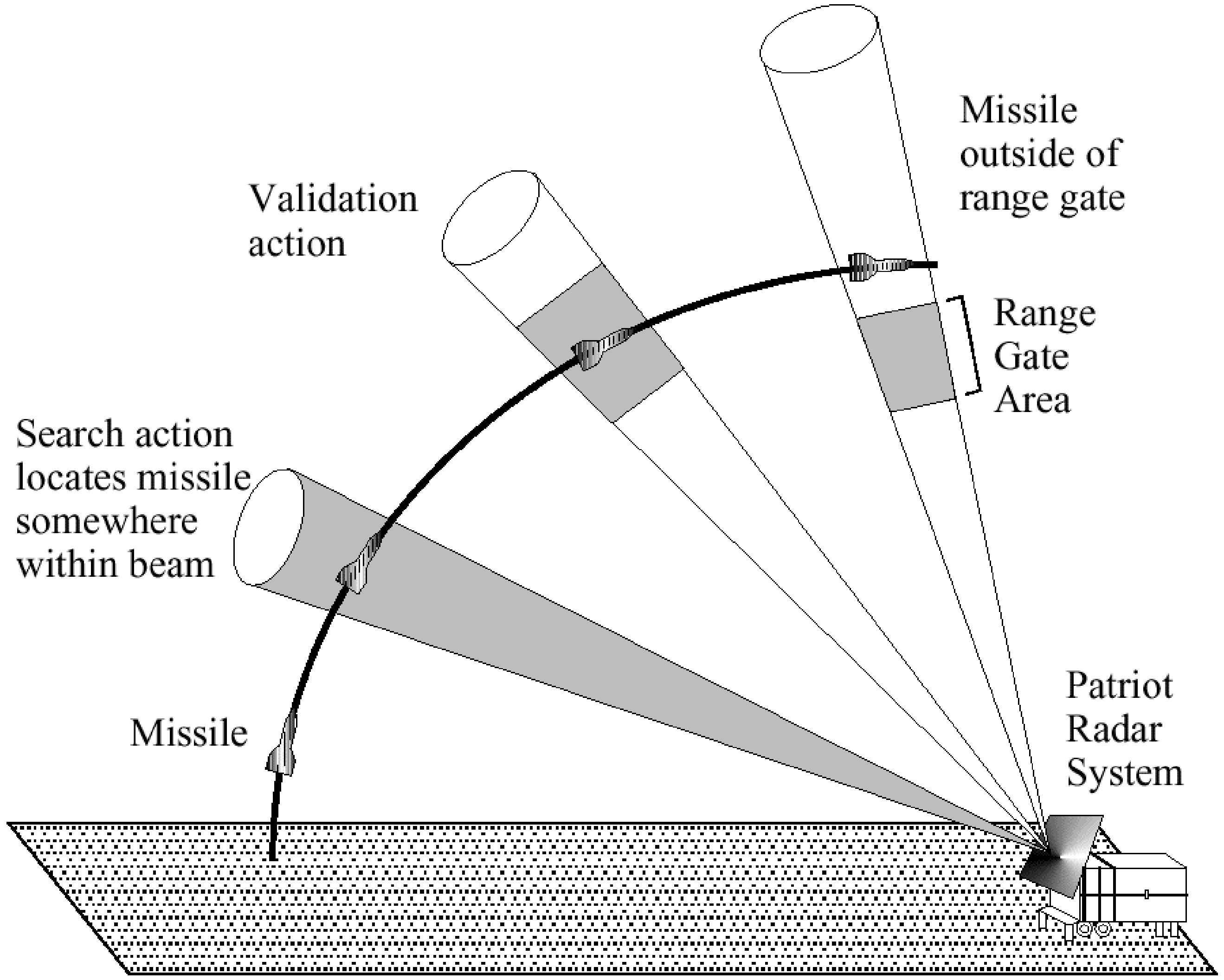
Evaluation of 1000*pi+e/20 as double, the result is converted to float

1.100 0100 0101 1011 1010 1000* 2^11 = 3141,728515625

Calculation performed by ***SpeedCrunch 0.12 - http://speedcrunch.org/***

Effect of Loss of Precision

According to the General Accounting Office of the U.S. Government, a loss of precision in converting 24-bit integers into 24-bit floating point numbers was responsible for the failure of a Patriot anti-missile battery.


Validation action
Missile outside of range gate
Range Gate Area
Search action locates missile somewhere within beam
Missile
Patriot Radar System

Slide source: UIUC

Effect of Loss of Precision

- During the Gulf War in 1991, a U.S. Patriot missile failed to intercept an Iraqi Scud missile, and 28 Americans were killed.
- A later study determined that the problem was caused by the inaccuracy of the binary representation of 0.10.
  - The Patriot incremented a counter once every 0.10 seconds.
  - It multiplied the counter value by 0.10 to compute the actual time.
- However, the (24-bit) binary representation of 0.10 actually corresponds to 0.099999904632568359375, which is off by 0.000000095367431640625.
- This doesn't seem like much, but after 100 hours the time ends up being off by 0.34 seconds—enough time for a Scud to travel 500 meters!
- UIUC Emeritus Professor Skeel wrote a short article about this.

Roundoff Error and the Patriot Missile. SIAM News, 25(4):11, July 1992.

Slide source: UIUC

How to add ?

We want to calculate the sum:

$$\sum_{i=1}^{N} \frac{1}{i^2}$$

More correct result is…

$$\sum_{i=1}^{N}\frac{1}{i^2}=\sum_{i=N}^{1}\frac{1}{i^2}=\sum_{i=1}^{N}\frac{1}{(N-i+1)^2}$$

$$\sum_{i=1}^{10^{10}}\frac{1}{i^2}\approx 1.6449340578301865$$

$$\sum_{i=10^{10}}^{1}\frac{1}{i^2}\approx 1.6449340668482264$$

added as type double...
Why are the results different? Select the more correct one.

Speed of real operations

Operation	Peformed by
Negation of number	negation of MSB (Main Scale Bit)
Comparison	a) sign-> b) absolute value
Multiply or divide by 2^n	change of exponent
Conversion among int, float, double	shift of mantissa according to exp.
Addition, Subtraction, Increment, decrement	Mantissas to the same exponents, +/-, rounding, normalization
Multiply by hardware multiplier	Add exponent, multiply mantissas, rounding, normalization
Multiply by sequence multiplier	
Division	Subtract exponents, divide mantissas, rounding, normlization

Iteration Divider Goldschmidt

$$Q = \frac{N}{B} = \frac{m_N 2^{e_N}}{m_B 2^{e_B}} = \frac{m_N}{m_B} 2^{e_N - e_b}$$

Normalized numbers:

m_N = 1.???????...?   a m_B = 1.???????...?

$1 \leq m_N, m_B < 2$ if we consider only mantissa, or
$0{,}5 \leq m_N, m_B < 1$ if we take in account only fractional part

Goldschmidt Division

- Let us compute the reciprocal of B (1/B)
  - Then, we can use the standard floating point multiplication algorithm
- Ignoring the exponent
  - Let us compute $(1/P_B)$, where P_B is mantissa
- If B is a normal floating point number
  - $1 <= P_B < 2$
  - $P_B = 1 + X$ where $(X < 1)$

Source: IIT Delhi, McGrawHill

Goldschmidt Division - II

$$
\begin{aligned}
\frac{1}{P_B} &= \frac{1}{1+X} \quad (P_B = 1 + X, 0 < X < 1) \\
&= \frac{1}{1 + 1 - X'} \quad (X' = 1 - X, X' < 1) \\
&= \frac{1}{2 - X'} \\
&= \frac{1}{2} * \frac{1}{1 - \frac{X'}{2}} \\
&= \frac{1}{2} * \frac{1}{1 - Y} \quad (Y = \frac{X'}{2} = (1 - X)/2, Y < \frac{1}{2})
\end{aligned}
$$

Source: IIT Delhi, McGrawHill

Goldschmidt Division - III

$$\begin{aligned}
\frac{1}{1-Y} &= \frac{1+Y}{1-Y^2} \\
&= \frac{(1+Y)(1+Y^2)}{1-Y^4} \\
&= \ldots \\
&= \frac{(1+Y)(1+Y^2)\ \ldots\ (1+Y^{16})}{1-Y^{32}} \\
&\approx (1+Y)(1+Y^2)\ \ldots\ (1+Y^{16})
\end{aligned}$$

* There is no point considering Y^{32} because it cannot be represented in our format!

Source: IIT Delhi, McGrawHill

Generating the 1/(1-Y)

$$(1+Y)(1+Y^2)\ \ldots\ (1+Y^{16})$$

- We can compute Y^2 using a FP multiplier.
  - Again square it to obtain Y^4, Y^8, and Y^{16}
  - Takes 4 multiplications, and 5 additions, to generate all the terms
  - Need 4 more multiplications to generate the final result (1/1-Y)
- Compute $1/P_B$ by a single right shift

Source: IIT Delhi, McGrawHill

*Program

for contemplation

Quiz: Deside if it is true

```
int x = …;
float f = …;
double d = …;
```

We suppose that f and d are not NaN

- `x == (int)(float) x`
- `x == (int)(double) x`
- `f == (float)(double) f`
- `d == (float) d`
- `f == -(-f);`
- `2/3 == 2/3.0`
- `d < 0.0 ⇒ ((d*2) < 0.0)`
- `d > f ⇒ -f > -d`
- `d * d >= 0.0`
- `(d+f)-d == f`

Answers

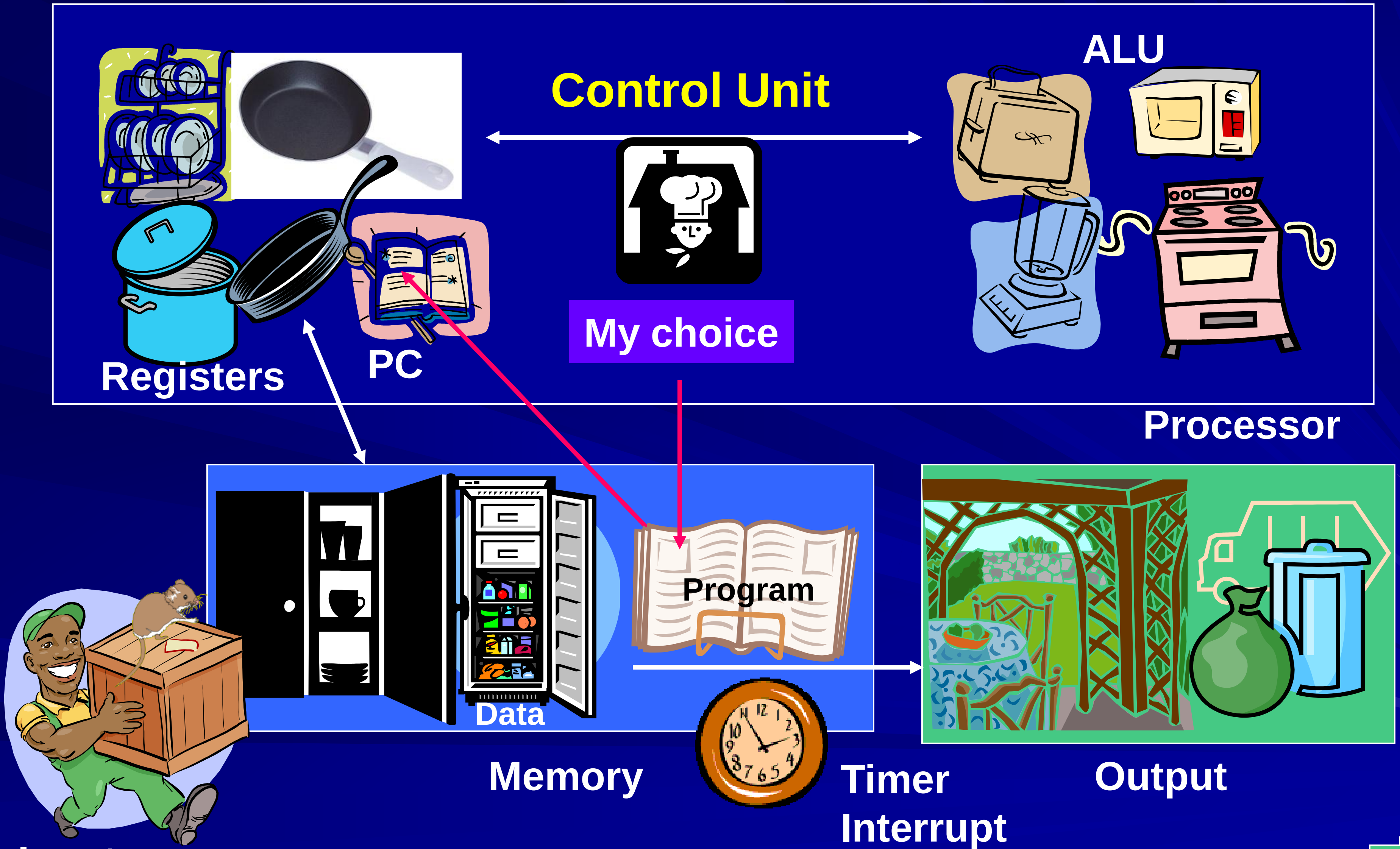
```
int x = …;

float f = …;

double d = …;
```

We suppose that f and d are not NaN

`x == (int)(float) x`	**No: 24 significant bits**
`x == (int)(double) x`	**Yes: 53 significant bits**
`f == (float)(double) f`	**Yes : precession is increased**
`d == (float) d`	**No: precession is lost**
`f == -(-f);`	**Yes: change of sign only**
`2/3 == 2/3.0`	**No: 2/3 == 0**
`d < 0.0 ⇒ ((d*2) < 0.0)`	**Yes!**
`d > f ⇒ -f > -d`	**Yes!**
`d * d >= 0.0`	**Yes!**
`(d+f)-d == f`	**No: Not associative**

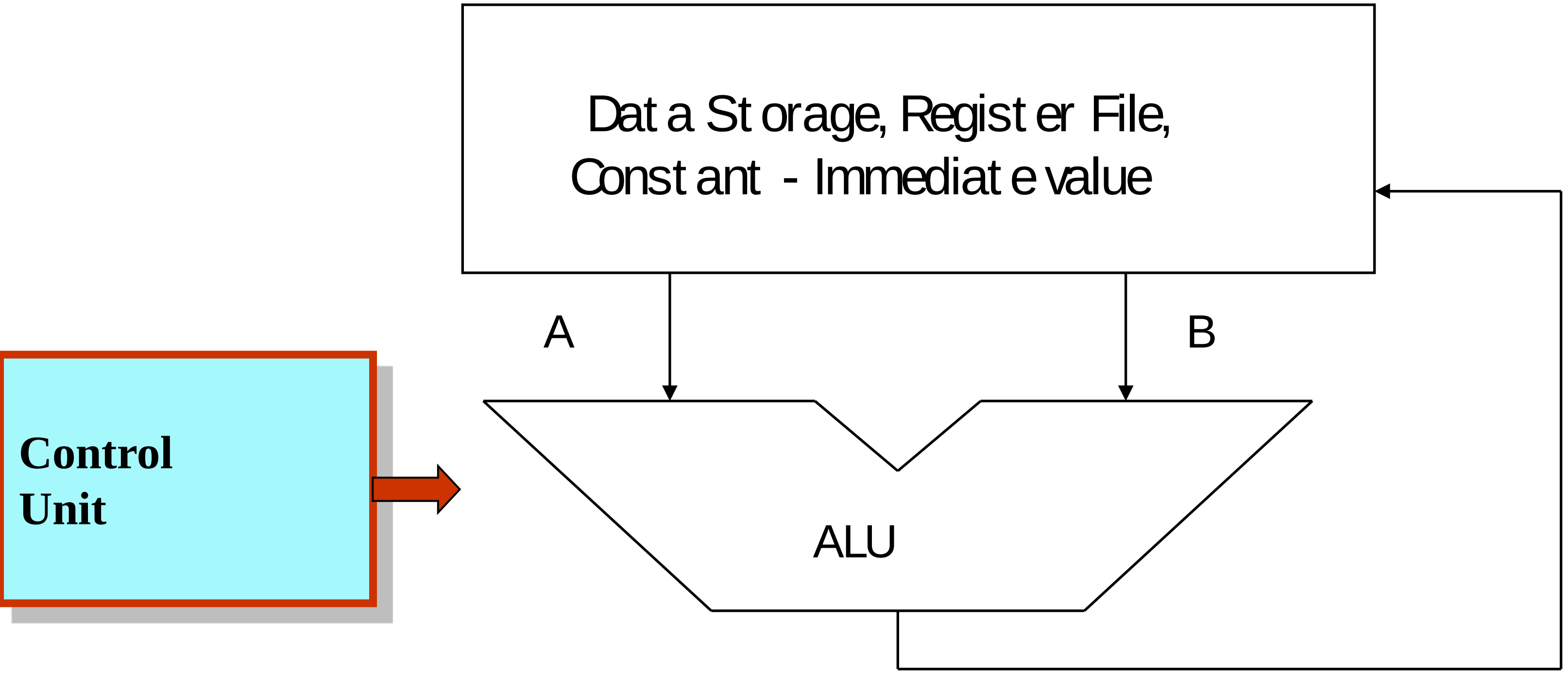
Microprocessors

Processor is something à la Kitchen



[Nelson: Computer Architecture and Design, Auburn 2008]

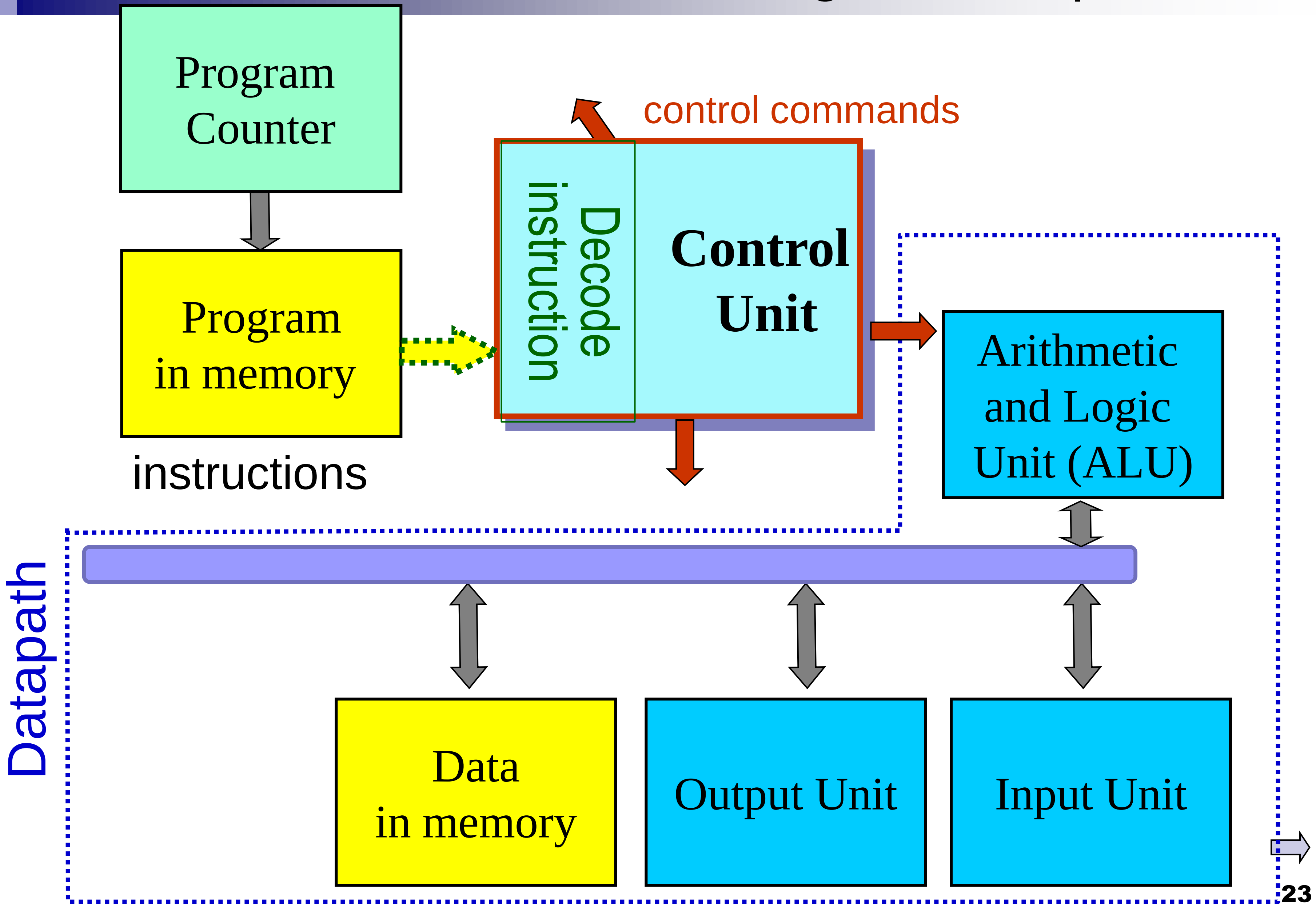
Simplified Processor



Control unit controls datapath

Inherently sequential.

Model of Digital Computer



RISC CPU Design Stategy

RISC - Reduced Instruction Set Computer

Its philosophy - keep it simple!

- **fixed instruction length**(s) (usually one word)
- **load-store** instruction sets (don’t do anything else)
- **limited addressing modes**
- **limited operations**

Examples: MIPS, Sun SPARC, HP PA-RISC, IBM PowerPC, Intel (Compaq), Alpha, NIOS…

Design goals:
speed, size, power consumption, reliability,
cost ← design, fabrication, test, packaging,
space on chip ← embedded systems

CISC Design Strategy

CISC = Complex Instructions Set Computers

Examples of CISC Instruction

Machine	Instruction	Effect
Pentium	`MOVS`	Move string of bytes, words, or double words
PowerPC	`cntlzd`	Count the number of consecutive 0s
IBM 360-370	`CS`	Compare and swap register if a condition is satisfied
Digital VAX	`POLYD`	Evaluation of polynomial using a coefficient table

Computer based on von Neumann's concept

- Control unit
- ALU
- Memory
- Input
- Output

Processor/microprocessor

von Neumann architecture uses common memory, whereas Harvard architecture uses separate program and data memories

Input/output subsystem

The control unit is responsible for control of the operation processing and sequencing. It consists of:

- **registers** – they hold intermediate and programmer visible state
- **control logic circuits** which represents core **of the control unit** (CU)

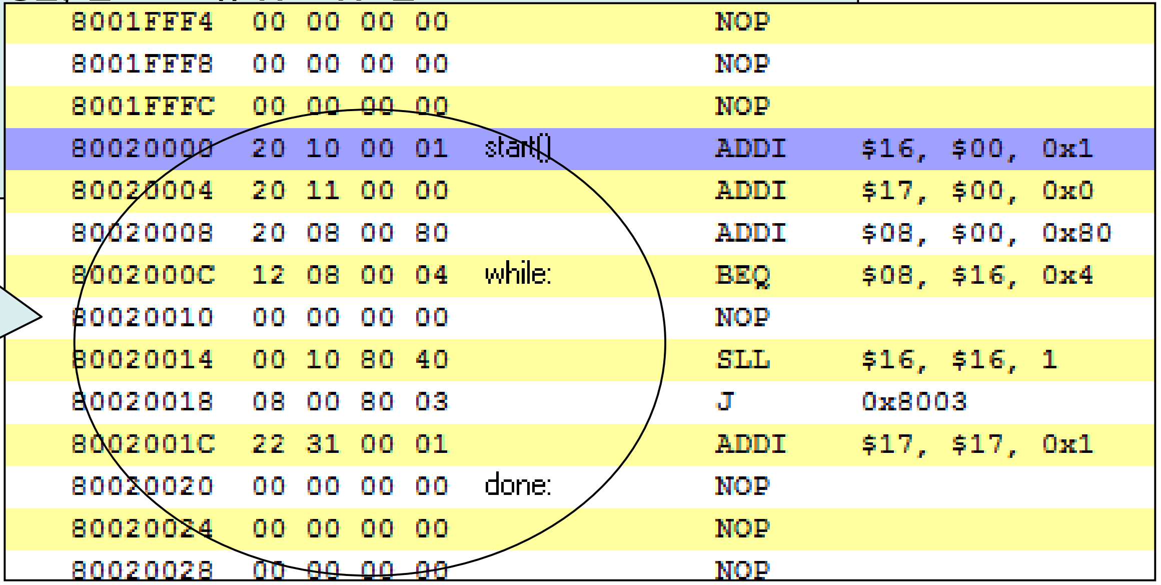
Registers

- **Assembly operands are registers**
  - registers are special memory elements inside CPU that allow fast access
  - operations can only be performed on them!
- **MIPS registers are 32 bit wide**
  - they are numbered from \$0 to \$31
  - Each register can be referred to by its number or defined name:
    - number references: \$0, \$1, \$2, … \$30, \$31
    - named references: zero, at, v0, ..., fp, ra

Compilation: C -> Assembler -> Machine Code

```
int pow = 1;
int x = 0;

while(pow != 128)
{
  pow = pow*2;
  x = x + 1;
}
```

```
addi s0, $0, 1        // pow = 1
addi s1, $0, 0        // x = 0
addi t0, $0, 128      // t0 = 128

while:
  beq  s0, t0, done  // if pow==128, go to done
  sll  s0, s0, 1      // pow = pow*2
  addi s1, s1, 1      // x = x+1
  j    while
done:
```



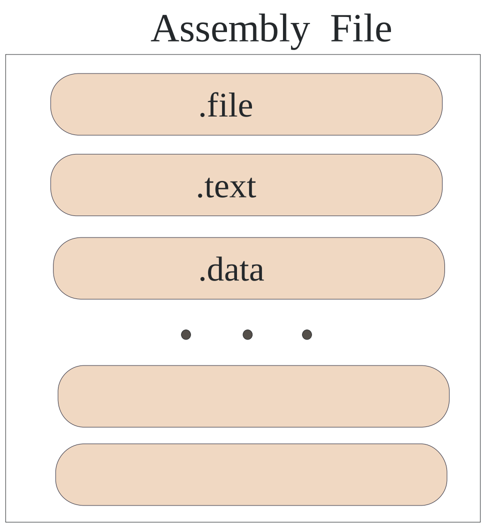
```
8001FFF4  00 00 00 00            NOP
8001FFF8  00 00 00 00            NOP
8001FFFC  00 00 00 00            NOP
80020000  20 10 00 01  start()   ADDI  $16, $00, 0x1
80020004  20 11 00 00            ADDI  $17, $00, 0x0
80020008  20 08 00 80            ADDI  $08, $00, 0x80
8002000C  12 08 00 04  while:    BEQ   $08, $16, 0x4
80020010  00 00 00 00            NOP
80020014  00 10 80 40            SLL   $16, $16, 1
80020018  08 00 80 03            J     0x8003
8002001C  22 31 00 01            ADDI  $17, $17, 0x1
80020020  00 00 00 00  done:     NOP
80020024  00 00 00 00            NOP
80020028  00 00 00 00            NOP
```

MIPS: Common Register Usage

Reg	Name	Normal usage
$0	**zero**	0x0000_0000 - read only!
$1	**at**	Assembler Temporary
$2	**v0**	Unsaved function arguments and return value
$3	**v1**	
$4	**a0**	
$5	**a1**	
$6	**a2**	
$7	**a3**	
$8	**t0**	Unsaved temporaries
$9	**t1**	
$10	**t2**	
$11	**t3**	
$12	**t4**	
$14	**t5**	
$14	**t6**	
$15	**t7**	

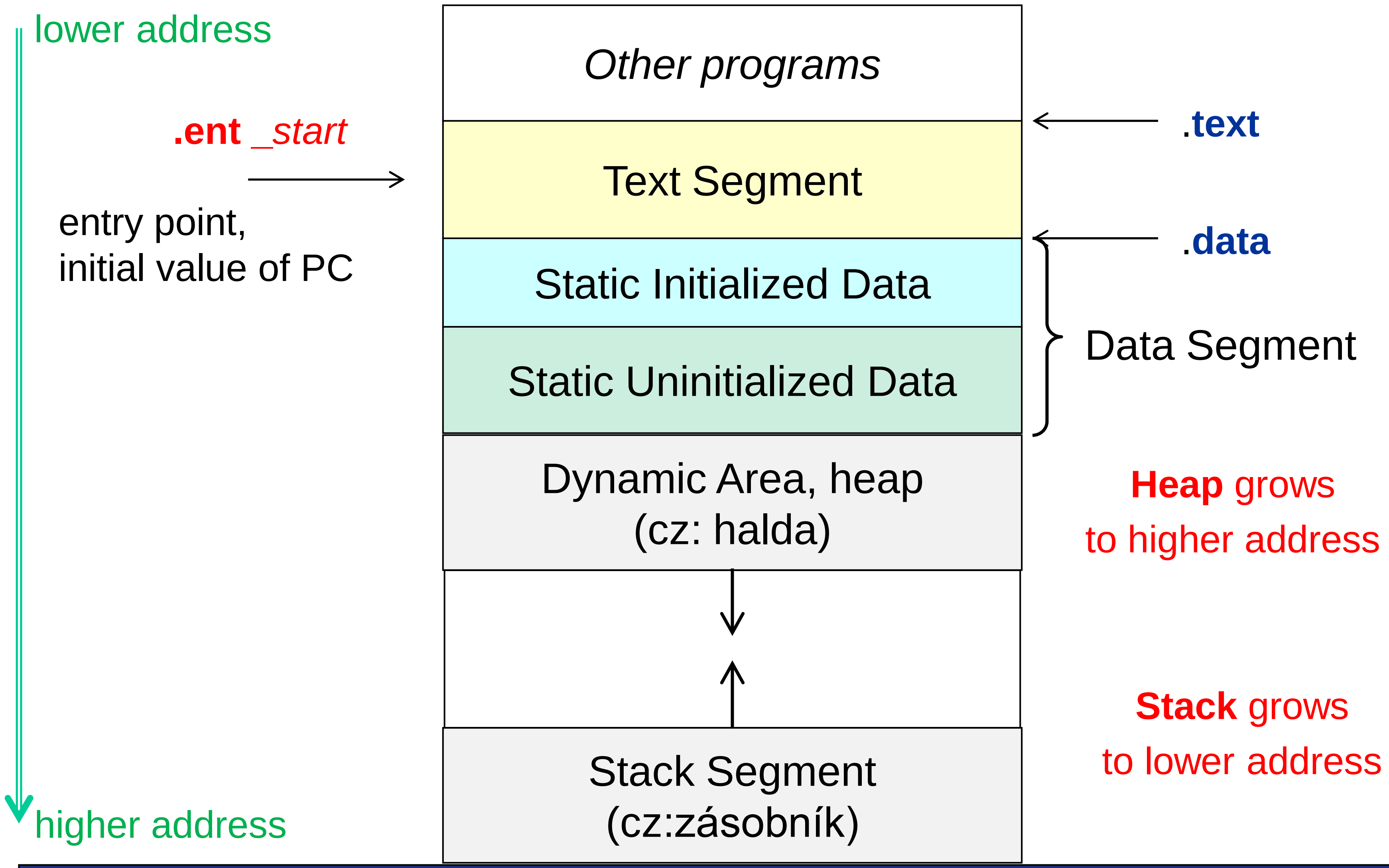
Reg	Name	Normal usage
$16	**s0**	Saved Temporaries
$17	**s1**	
$18	**s2**	
$19	**s3**	
$20	**s4**	
$21	**s5**	
$22	**s6**	
$23	**s7**	
$24	**t8**	
$25	**t9**	
$26	**k0**	Interrupt
$27	**k1**	Reserved
$28	**gp**	Global Pointer
$29	**sp**	Stack Pointer
$30	**fp**	Frame Pointer
$31	**ra**	return Address

Assembly File



- Divided into different sections
- Each section contains some data, or assembly instructions

Layout of a Program in Memory



Assembly code - preview

```
/* template for own QtMips program development */
.globl _start    // .globl makes the symbol visible to linker
.set noat         // disables warning when $at register is used by user.
.set noreorder // prevents the assembler from reordering machine-language instructions
                     // See later lectures
.ent _start
.text
_start:
        lw $2, 0x2000($0)     // load the word from absolute address
        sw $2, 0x2004($0) // store the word to absolute address

loop:
        break // stop execution wait for debugger/user
        beq $0, $0, loop // endless loop
        // it ensures that continuation does interpret random data
        nop
.data
src_val:
        .word 0x12345678
dst_val:
.end _start
```

Assembly code

- Three types of statements in assembly language:
  - Typically, one statement should appear on a line

1. Executable Instructions
   - Generate machine code for the processor to execute at runtime
   - Instructions tell the processor what to do
2. Pseudo-Instructions and Macros
   - Translated by the assembler into real instructions
   - Simplify the programmer task
3. Assembler Directives
   - Provide information to the assembler while translating a program
   - Used to define segments, allocate memory variables, etc.
   - Non-executable: directives are not part of the instruction set

.data directive - Definition Directives

*Sets aside storage in memory for a variable and optionally assigns a **name** (label) to the data*

Syntax:

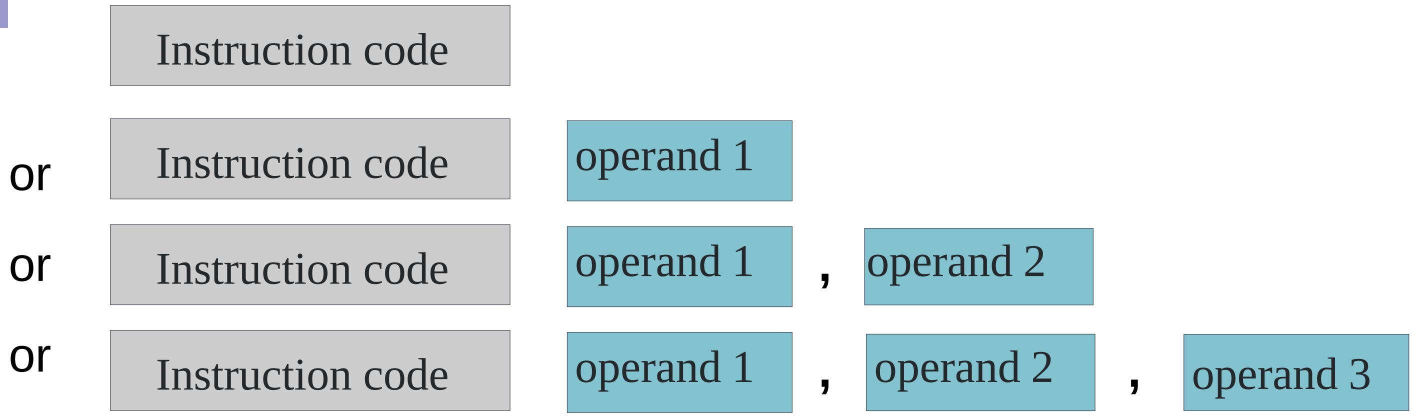
[***name***:] *directive* *initializer* [, *initializer*] . . .

```
var1:    .word  10
myarray: .word  5, 3, 4, 1, 15
```

All initializers become binary data in the initialized memory, we will discuss this topic more in the next lecture. The location of the text and data sections can be specified by compiler parameters, e.g.

```
mips-elf-gcc -Wl,-Ttext,0x1000 -Wl,-Tdata,0x2000 -nostdlib -nodefaultlibs -
                nostartfiles -o simple-lw-sw simple-lw-sw.S
```

Structure of Instructions


Instruction code
or Instruction code operand 1
or Instruction code operand 1 , operand 2
or Instruction code operand 1 , operand 2 , operand 3

- instruction textual identifier of a machine instruction
- operands
  - register
  - memory location
  - constant (also known as an immediate)

Instruction Formats

All instructions are **32-bit wide**.

Register (R-Type)

Register-to-register instructions, Rx are numbers of registers
Op: operation code specifies the format of the instruction,
funct- sub-function, control codes
sa - used with the shift and rotate instructions,

Op^6	Rs^5	Rt^5	Rd^5	sa^5	funct^6

Immediate (I-Type)

16-bit immediate constant is a part of the instruction

Op^6	Rs^5	Rt^5	immediate^{16}

Jump (J-Type)

Used by jump instructions only

Op^6	immediate^{26}

Upper indexes specify bit widths of fields in an instruction.

ALU Instructions

operation	R-format	I-format
add	`add` `addu`	`addi` `addiu`
subtract	`sub`	–

multiply
divide
	`mult / multu`
	`div / divu`
		–

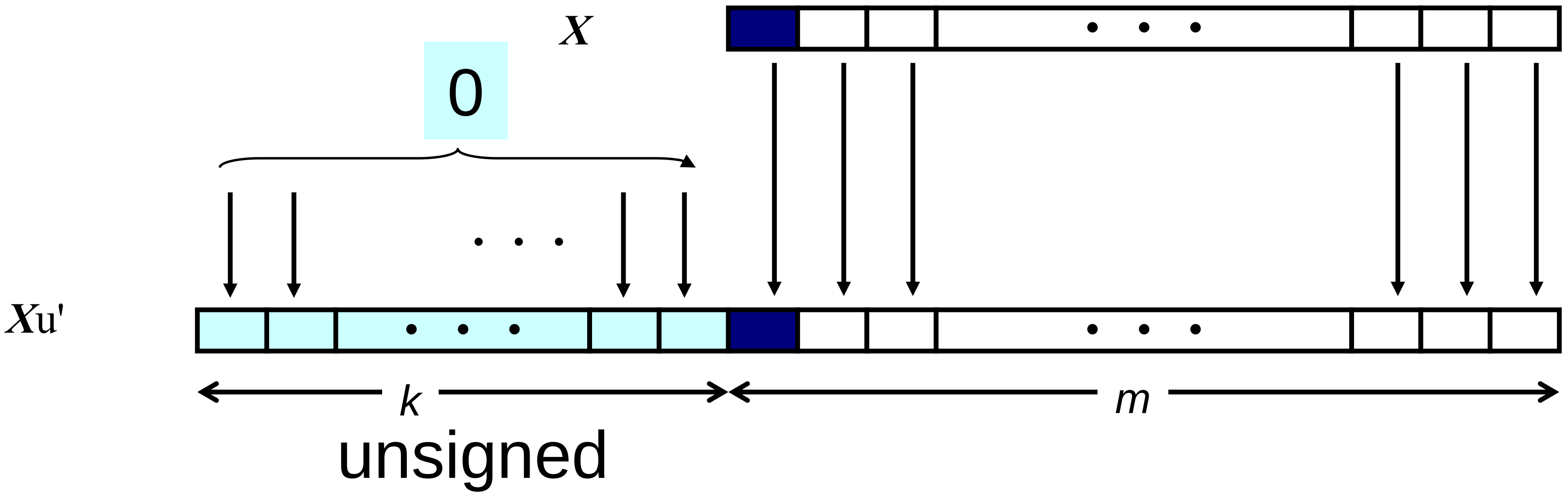
AND	`and`	`andi`
OR	`or`	`ori`
XOR	`xor`	`xori`
NOR	`nor`	–

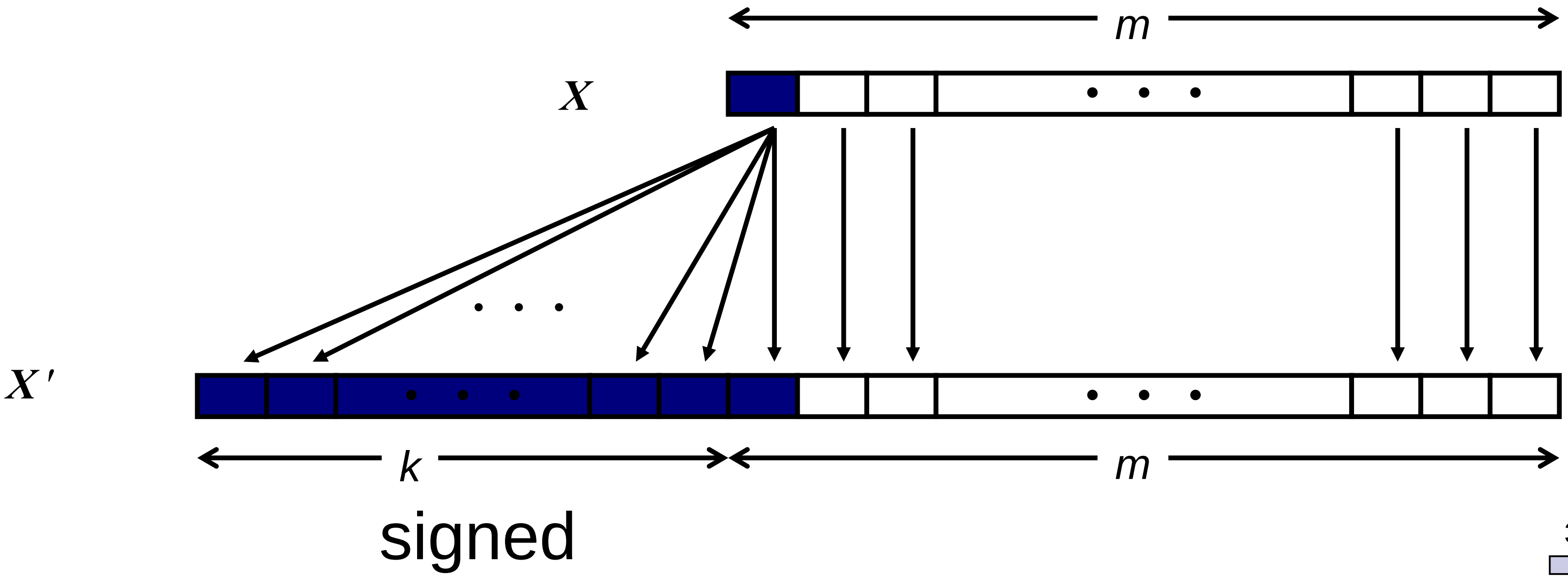
addi, addiu $\mathbf{rB \leftarrow rA + se(\mathit{number})}$,

addu, addiu - no overflow trap

Logic instructions AND, OR, XOR, NOR do not use se = sign-extension

Unsigned/Signed Extension





The Constant Zero

- MIPS register $0 (**zero**) is the constant 0
  - Cannot be overwritten!
- Useful for common operations
  - E.g., move between registers

```
add $8, $9, $zero
```

$8 ← $9

How to load a value int register ?

ori $1, $0, 1000	$1← 1000
addi $2, $0, 1000	$2 ← 1000
lui $3, 0x1234	$3 ← 0x12345678
ori $3, 0x5678	
la $3, 0x12345678	la - pseudo-instrukce

Instr.	Syntax	Operace
Load upper immediate: The immediate value C is shifted left 16 bits and stored in the register. The lower 16 bits of the register are zeroes.		
lui	lui $t,C	$t = C << 16
Load Address: The 32-bit label is stored into the $r register. This is a pseudoinstruction - it is translated by other instructions.		
la	la $r, LabelAddr	lui $r, LabelAddr[31:16]; ori $r,$r, LabelAddr[15:0]

Some shift operations

Instr.	Syntax	Operace	Význam
sll	sll \$d,\$s,C	\$d = \$s << C	***Shift Logical Left***: *Shifts register \$s left by C bits and places the result in \$d. Zeroes are shifted in. (equivalent to multiplying by* 2^C*)*
srl	srl \$d,\$s,C	\$d = \$s >> C *unsigned*	***Shift Logical Right***:*Shifts register \$s right by C bits and places the result in \$d.* ***Zeroes are shifted in.*** *(equivalent to dividing by* 2^C*)*
sra	sra \$d,\$s,C	\$d = \$s >> C *signed*	***Shift Arithmetic Right***: *Shifts register \$s right by C bits and places the result in \$d.* ***The sign bit is shifted in.*** *(equivalent to dividing by* 2^C*)*
nop	nop	sll \$0,\$0,0	*pseudoinstruction - no operation*

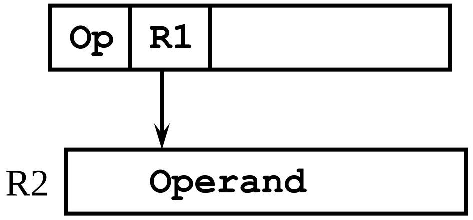
NOP binary code

```
000000 00000  00000 00000  00000 000000    -- fields of the instruction
opcode    $0        $0        0        funct              -- meaning of the fields
sll          source  dest   shft        sll
```

MIPS Addressing Modes

(a) Register direct addressing

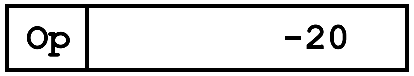
Register contains the operand



`or $1, $2, $3`

(b) Immediate addressing

Instruction contains the operand



`addi $1, $2, -20`

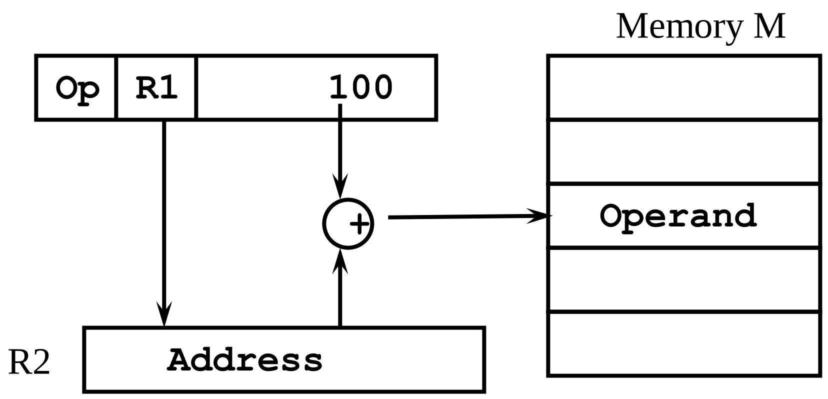
Memory Addressing

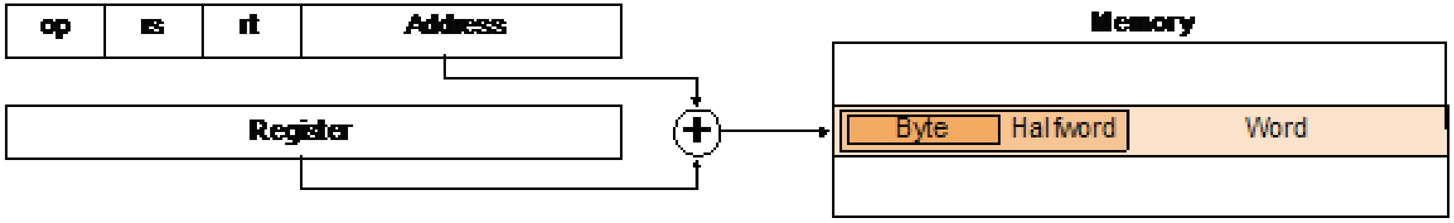
- (c) Displacement (or offset) addressing, it is also called base addressing
  - Address of operand = register + constant

  Memory address in load and store instructions is specified by a base register and offset

`sw R1, byte_offset(R2)`

sw $1, 100($2) : **$1** → Memory[**$2**+100]





Some of MIPS Memory Instruction

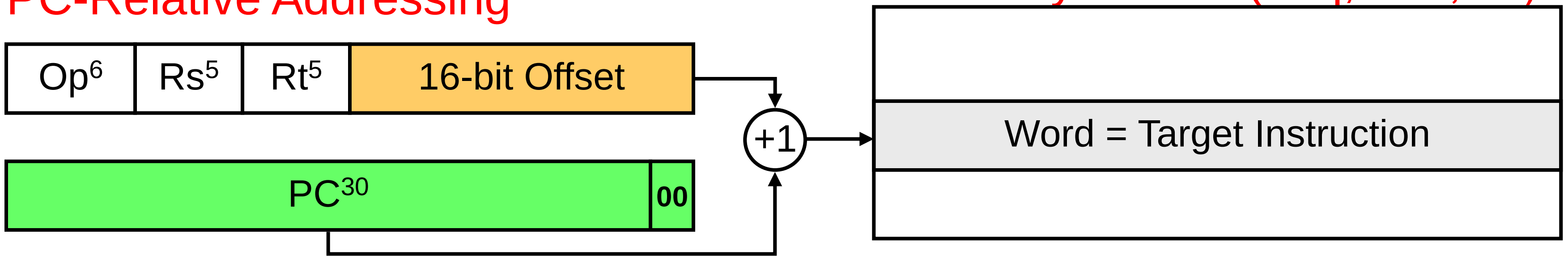
Instr.	Syntax	Operation	Performed as
lw	lw $t,C($s)	$t = Memory[$s + C]	**Load word:** A word is loaded into a register $t from the specified address.
sw	sw $t,C($s)	Memory[$s + C] = $t	**Store word**: The contents of $t is stored at the specified address.

We will discuss this topic more in the next lecture.

MIPS Jumps

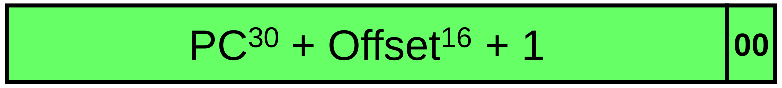
PC-Relative Addressing

Used by branch (beq, bne, …)



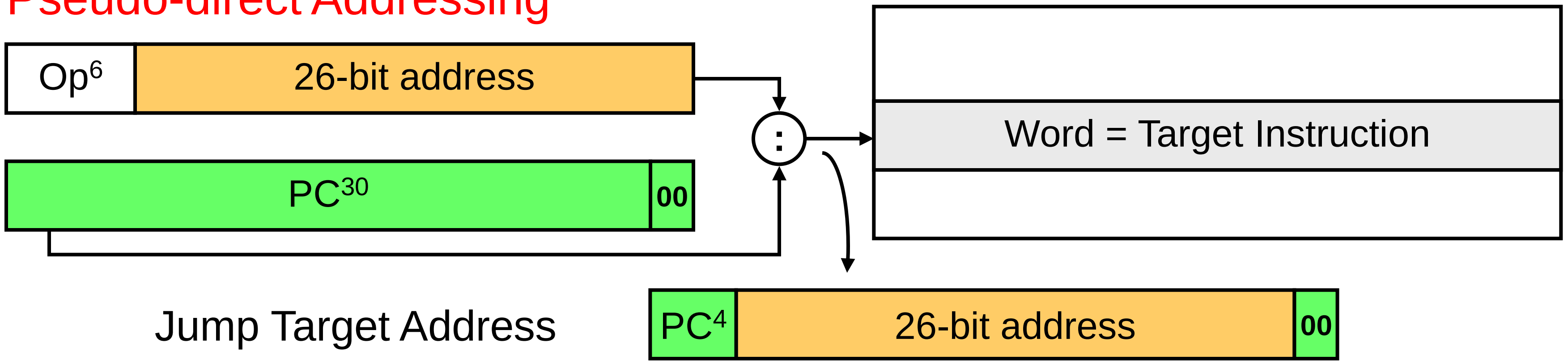
Branch Target Address

$$PC = PC + 4 \times (1 + Offset)$$
$$= PC+4+4*offset$$



Pseudo-direct Addressing

Used by jump instruction



Jump Target Address

Source: Dr. M. Mudawar, COE 301, KFUPM

MIPS Jump Instruction

Instrukce	**Syntax**	**Operace**
Branch on not equal: *Branches (jumps) if registers $s and $t are not equal*		

bne	bne $s, $t, offset	if $s != $t goto PC+4+4*offset;
		else goto PC+4

Branch on equal: : *Branches (jumps) if registers $s and $t are not equal*		

beq	beq $s, $t, offset	if $s == $t goto PC+4+4*offset;
		else goto PC+4

Jump: *Jumps always to label C*		
jump	j C	

MIPS Jump Instruction

Instrukce	Syntax	Operace
Set on less than: *If $s is less than $t, $d is set to 1. It gets zero otherwise. $s<imm as signed*		
slt, slti	slt $d,$s,$t slti $d, $s, imm	$d = ($s < $t) $d = ($s < imm)
Set on less than: *If $s is less than $t, $d is set to 1. It gets zero otherwise. $s<imm as unsigned*		
sltu, sltiu	sltu $d,$s,$t sltu $d, $s, imm	$d = ($s < $t) $d = ($s < imm)

Our assembler code - again

```
/* template for own QtMips program development */
.globl _start    // .globl makes the symbol visible to linker
.set noat         // disables warning when $at register is used by user.
.set noreorder // prevents the assembler from reordering machine-language instructions
                  // See later lectures
.ent _start
.text
_start:
      lw $2, 0x2000($0)     // load the word from absolute address
      sw $2, 0x2004($0) // store the word to absolute address

loop:
      break      // stop execution wait for debugger/user
      beq $0, $0, loop // endless loop
      // it ensures that continuation does interpret random data
      nop
.data
src_val:
      .word 0x12345678
dst_val:
.end _start
```