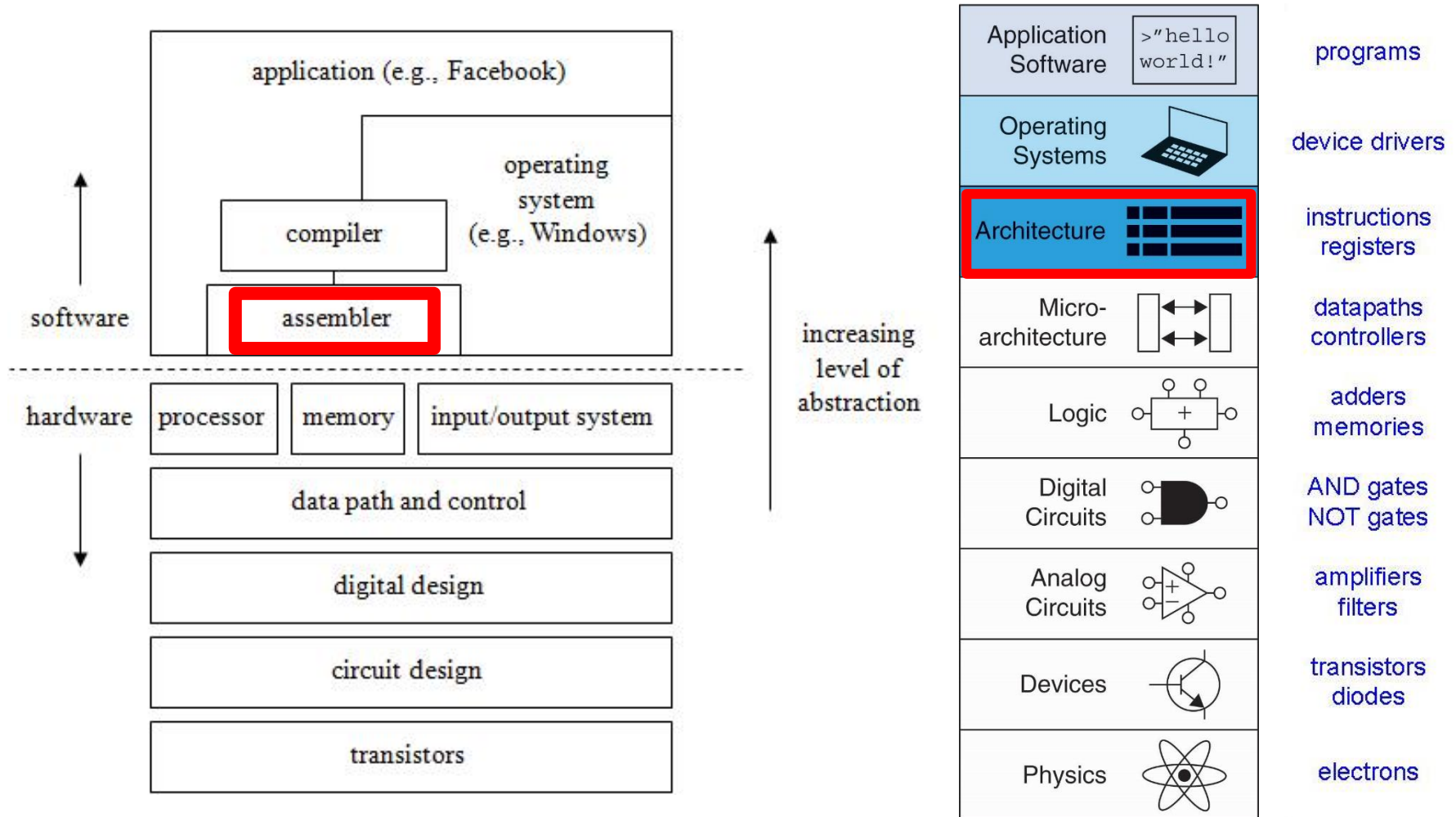


Computer Architecture

CISC/RISC

Computer levels



Addition:: Binary numbers

To add 2 binary numbers (**1+0**) ... Let us use ...

My *Add(2)* Computer System

My 2 binary numbers (**1**, **0**) are in the RAM...

My *Add(2)* Computer System

- STEP-1: Load (*ld*) from RAM memory to register *\$t0*, the binary: **1**

\$t0:

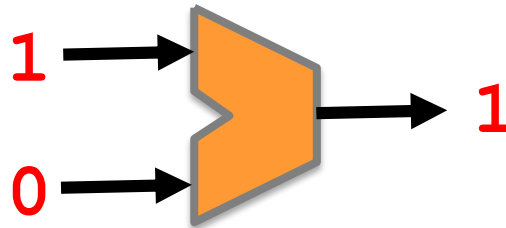
0	0	...	0	0	0	0	1
---	---	-----	---	---	---	---	----------

- STEP-2: Load (*ld*) from RAM memory to register *\$t1*, the binary: **0**

\$t1:

0	0	...	0	0	0	0	0
---	---	-----	---	---	---	---	----------

- STEP-3: *add* the two numbers (Arithmetic Logic Unit)



- STEP-4: Place the result to register *\$t2* ...

\$t2:

0	0	...	0	0	0	0	1
---	---	-----	---	---	---	---	----------

My Register-File

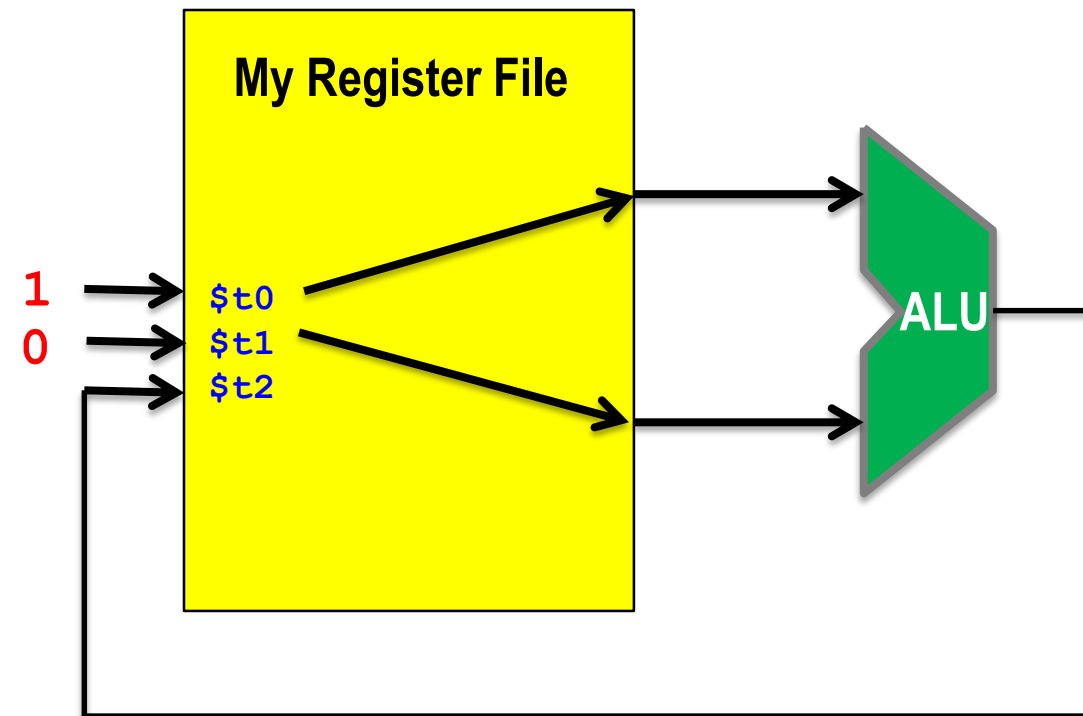
0	0	...	0	0	0	0	1	\$t0
0	0	...	0	0	0	0	0	\$t1
0	0	...	0	0	0	0	1	\$t2

...

...

...

My *Add(2)* Computer System



```
add $t2,$t1,$t0  
add rd, rs, rt  
add 1, 0, 1
```

rs (First **s**ource **r**egister operand)
rt (Second source **r**egister operand)
rd (**d**estination **r**egister operand)

Our first assembly instruction...

```
add rd, rs, rt
```

... and an example:

```
add $t2,$t1,$t0
```

rs (First **s**ource **r**egister operand)
rt (Second source **r**egister operand)
rd (**d**estination **r**egister operand)

We have ...

- Assembly Instruction: `add $t2, $t1, $t0`
- ALU
 - `ADD` 2 numbers ...
- Memory
 - Load from Memory to Register-File (`ld`) and back (`st`)
- Registers (3)
 - Store temporarily: `$t0`, `$t1`, `$t2`
 - How many registers do we have in a computer system?

Computer systems [CPU] architectures

- **CISC** (**C**omplex **I**nstruction **S**et **C**omputer)
 - 16 registers (**INTEL x86**, ...)
- **RISC** (**R**educed **I**nstruction **S**et **C**omputer)
 - 32 general registers (**MIPS**, **ARM**, **RISC-V**, ...)



CISC and RISC computers

	CISC	
	IBM370	80486
Instructions	208	235
Instruction size (bytes)	2-6	1-12
General registers	16	8
Developed	1973	1989

First CISC – IBM/360 (1964)

CISC and RISC computers

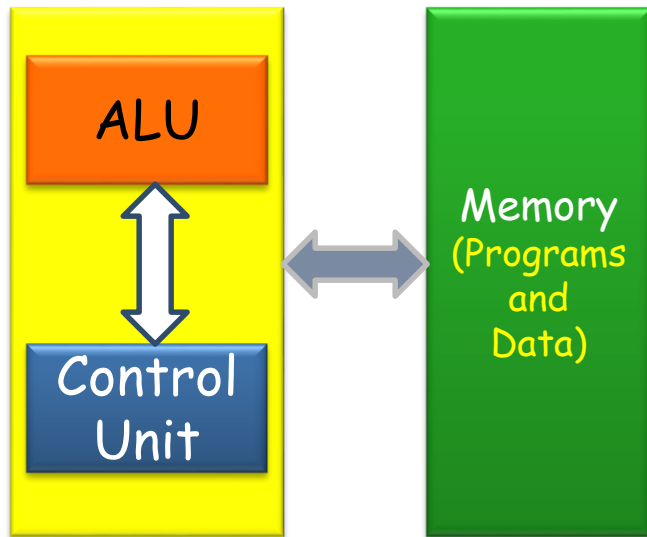
	CISC		RISC	
	IBM370	80486	SPARC	MIPS
Instructions	208	235	69	94
Instruction size (bytes)	2-6	1-12	4	4
General registers	16	8	40-580	32
Developed	1973	1989	1987	1991

First CISC – IBM/360 (1964)

First RISC – Burroughs B5000 (1963)

Von Neumann/Turing

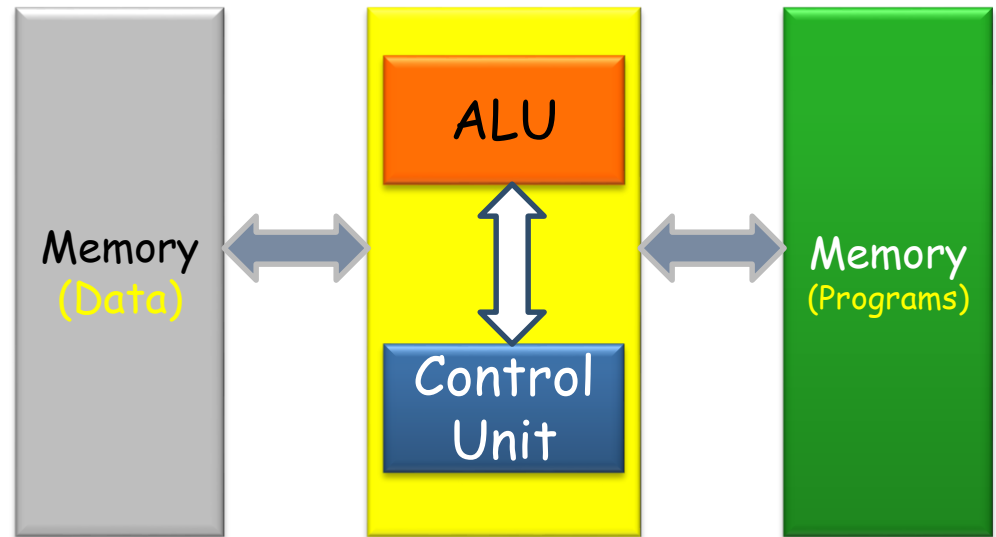
CISC



The von Neumann Architecture is named after the scientist John von Neumann (1942-1951)

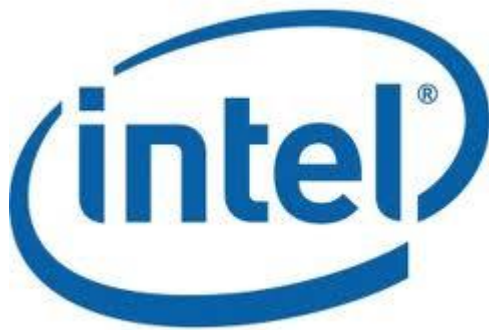
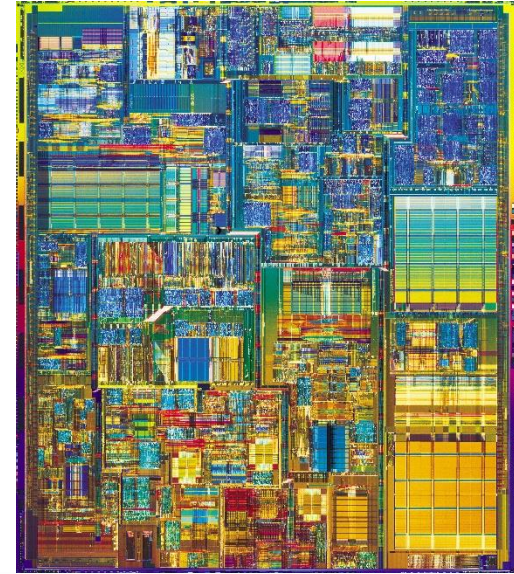
Harvard Architecture

RISC

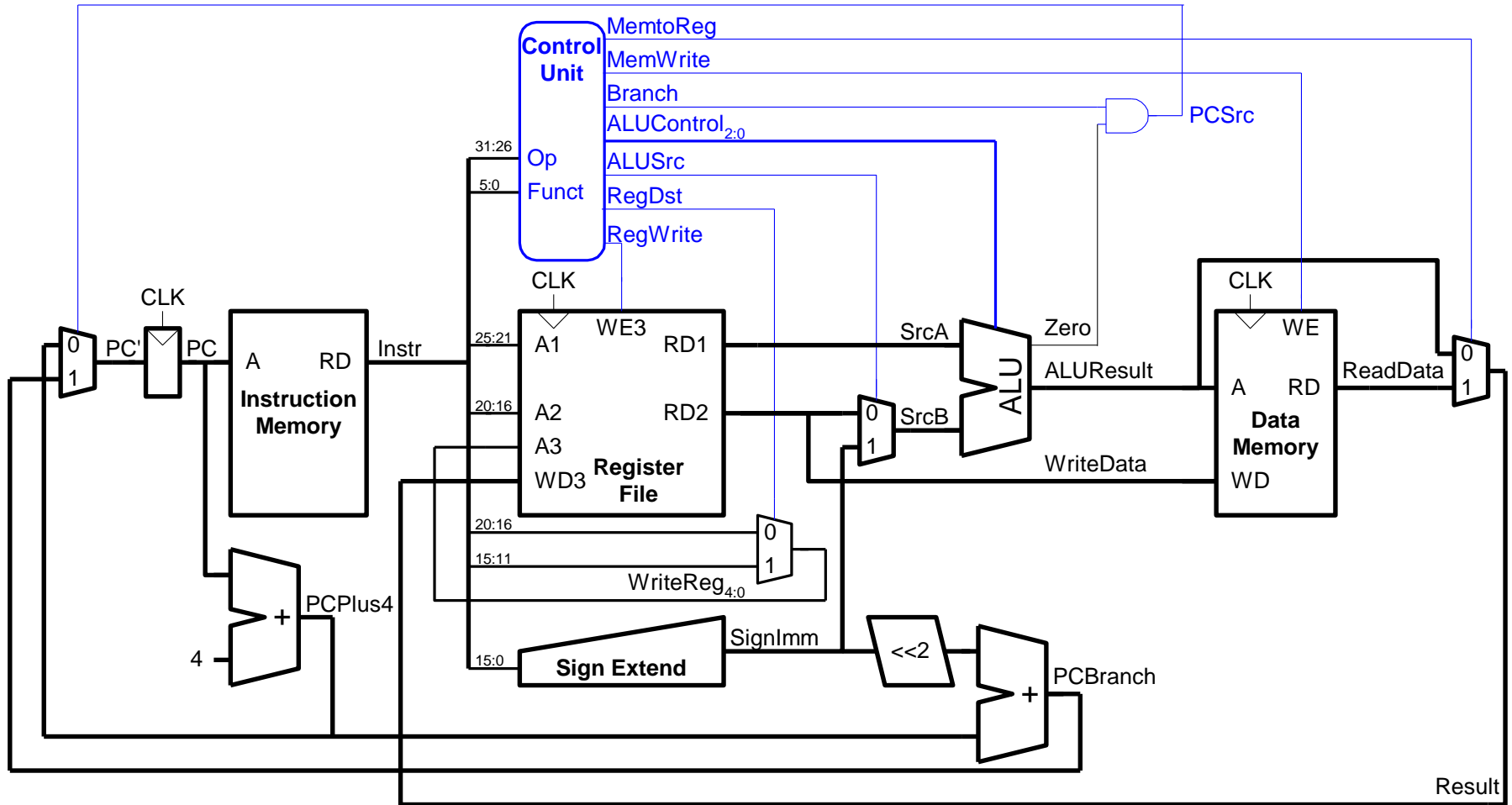


The name Harvard Architecture comes from the *Harvard Mark I* computer (1944)

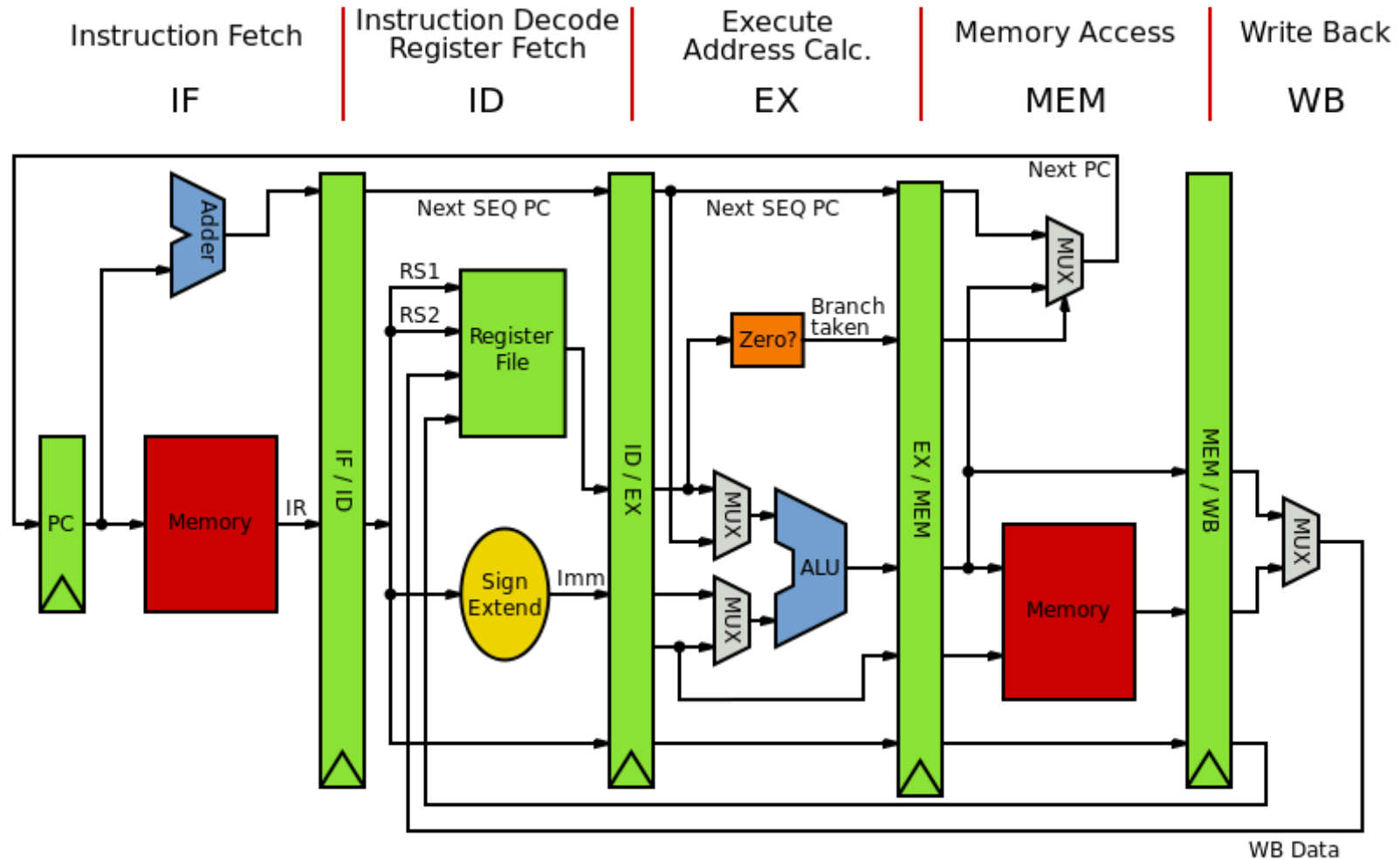
CISC (CPUs)



MIPS (RISC) Architecture [Single Cycled]



The MIPS – RISC Microprocessor



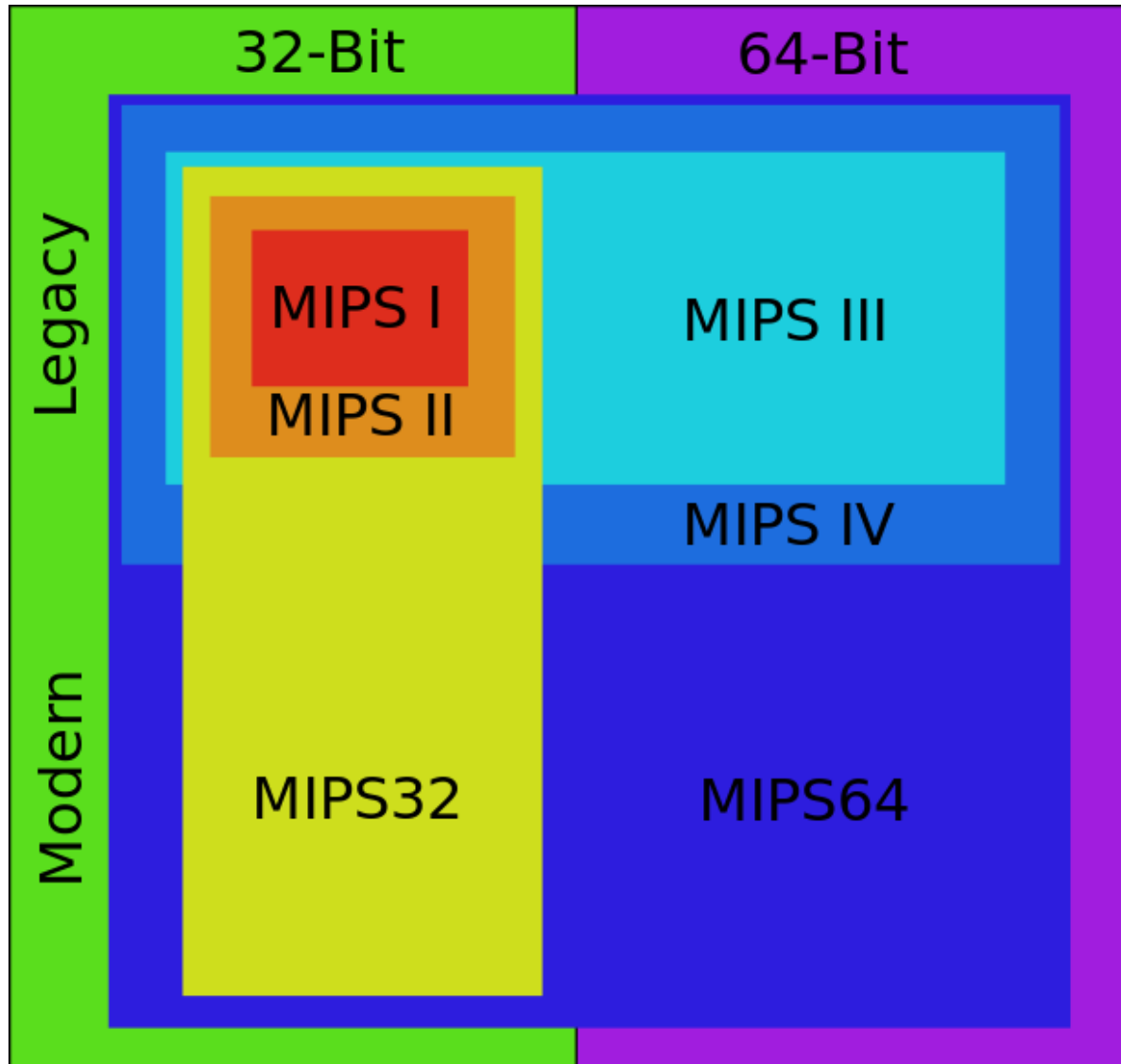
MIPS (**M**icroprocessor without **I**nterlocked **P**ipeline **S**tages) is a **R**educed **I**nstruction **S**et **C**omputer (**RISC**)

MIPS - RISC [32-bit] Registers

Name	Number	Use	Preserved across a call?
\$zero	0	The constant value 0	N.A.
\$at	1	Assembler temporary	No
\$v0-\$v1	2-3	Values for function results and expression evaluation	No
\$a0-\$a3	4-7	Arguments	No
\$t0-\$t7	8-15	Temporaries	No
\$s0-\$s7	16-23	Saved temporaries	Yes
\$t8-\$t9	24-25	Temporaries	No
\$k0-\$k1	26-27	Reserved for OS kernel	No
\$gp	28	Global pointer	Yes
\$sp	29	Stack pointer	Yes
\$fp	30	Frame pointer	Yes
\$ra	31	Return address	Yes

Figure 1.4 MIPS registers and usage conventions. In addition to the 32 general-purpose registers (R0-R31), MIPS has 32 floating-point registers (F0-F31) that can hold either a 32-bit single-precision number or a 64-bit double-precision number.

MIPS32 and MIPS64 - RISC



'D' = double-word (64 bits)

'W' = word (32 bits)

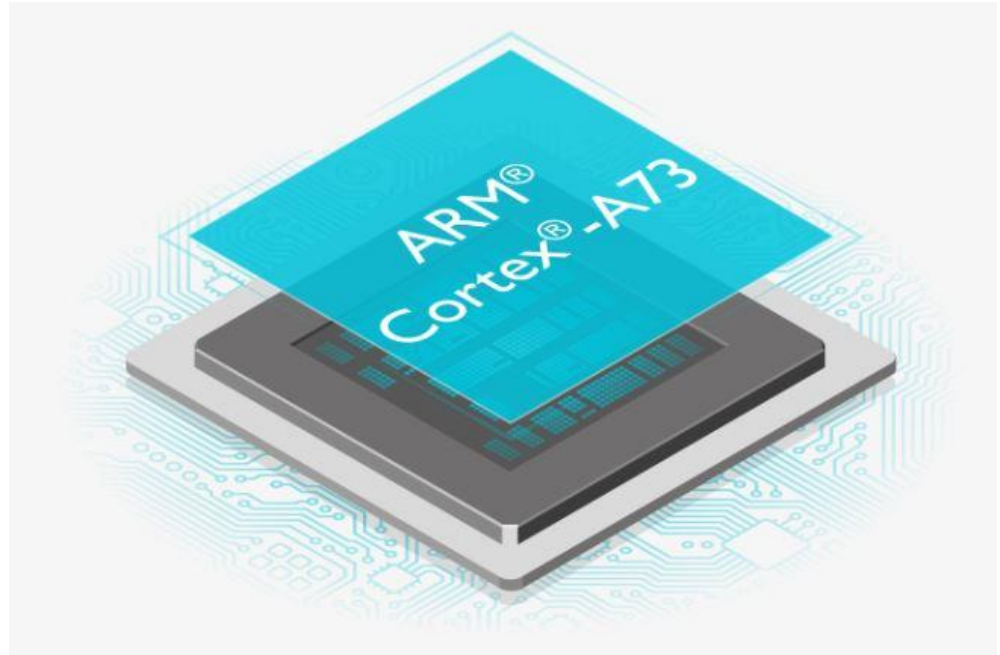
'H' = half-word (16 bits)

'B' = byte (8-bits)

RISC (CPUs)



iPhone 3G S

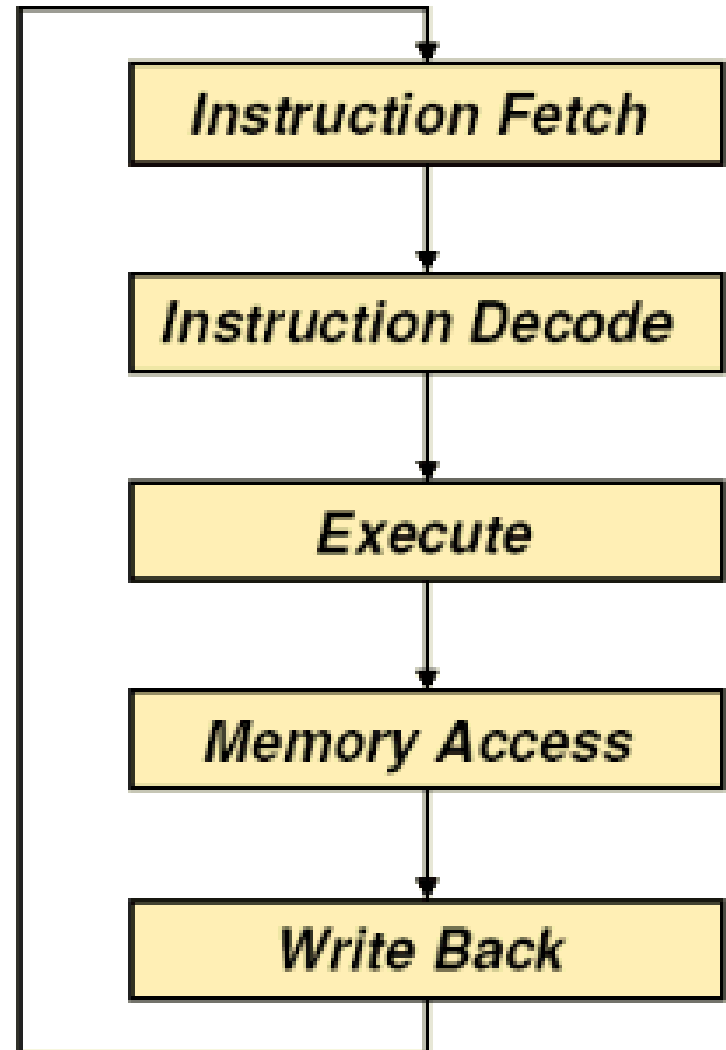


add \$t2, \$t1, \$t0

MIPS Assembly Instruction

Each MIPS Assembly Instruction ...

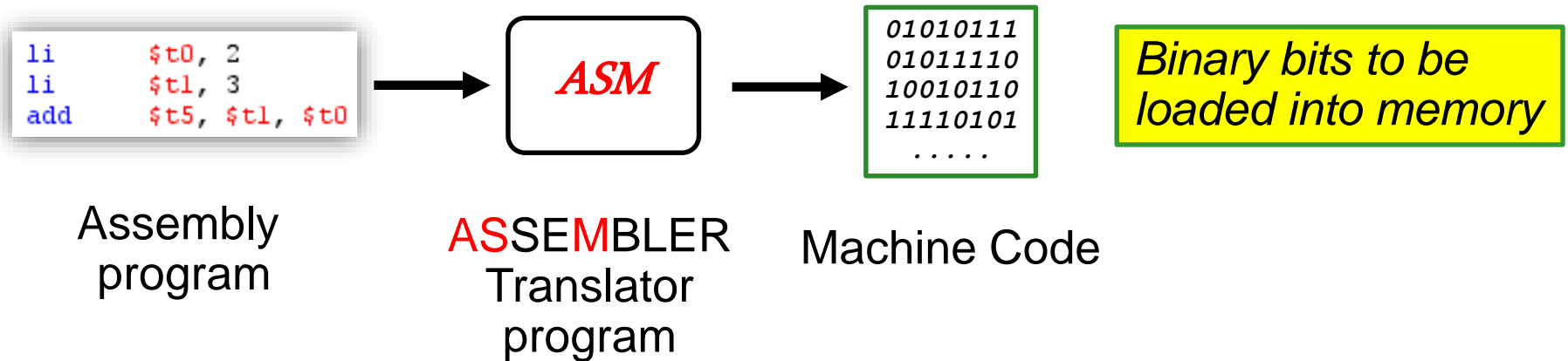
```
ld $t0, (from)
add $t2, $t1, $t0
st $t2, (there)
```



Next Assembly Instruction ...

- The CPU reads **the next instruction** from memory
 - It is placed in the **I**nstruction **R**egister (**IR**)
- The **P**rogram **C**ounter (**PC**) register holds the address of **the current Instruction**.
- The control unit then **translates the instruction into the sequence of micro-operations** necessary to implement the instruction.

Assembly..[Assembler]..Machine language



Assembly programming

Machine Code

0000.0001.0010.1001.0100.0000.0010.000

English

“Put sum of `$t0` and `$t1` into `$t2`”

- Assembly language is a compromise
 - Programmers don't like machine code
 - Computers can't understand English
- Mnemonics = symbolic names for instructions

Instruction

`ld $t0, (from)`

`add $t2, $t1, $t0`

`st $t2, (there)`

Description

Load the word at address `from` into `$t0`

Put the sum of `$t0` and `$t1`, into `$t2`

Store the word in `$t2`, into address `there`

Groups of instructions [MIPS32 Assembly]

- Arithmetic [Add/Subtract, Multiply, Divide ...]
- Logic [AND, OR, NOR]
- Data [Move, Load, Store, ...]
- Control flow [JUMP, ...].

Multiply 2 numbers: CISC and RISC

Let's say we want to find the product of two numbers...

- One number is stored in location [2:3] and another stored in location [5:2]
- Then store the product back in the location [2:3].

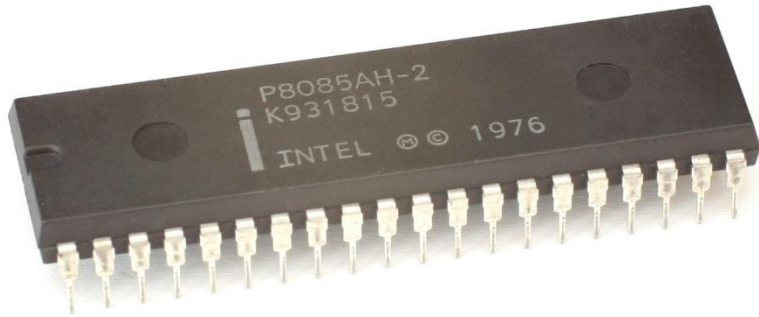
CISC

MULT [2:3], [5:2]

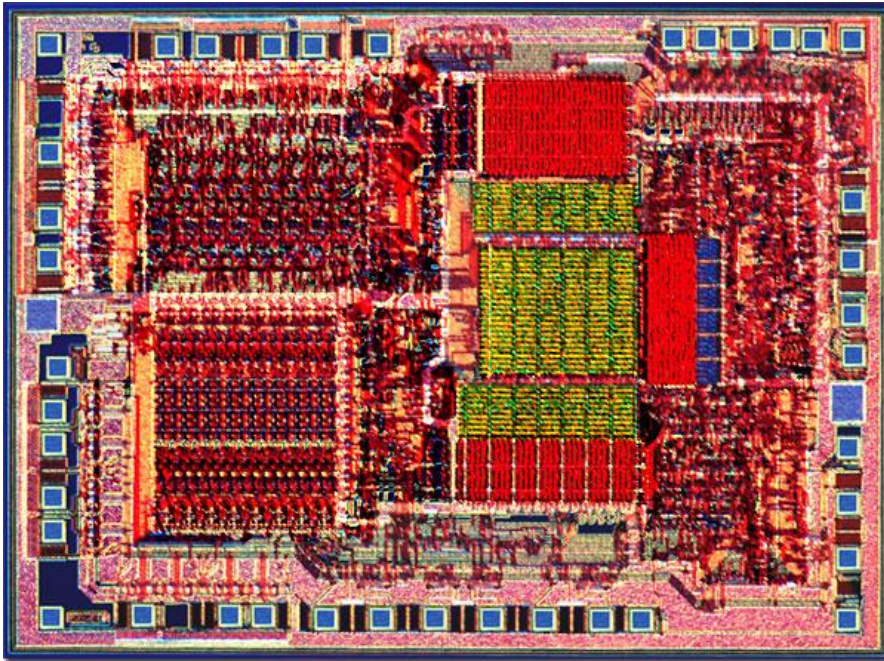
RISC

```
LOAD  A, [2:3]
LOAD  B, [5:2]
MULT  A, B
STORE [2:3], A
```

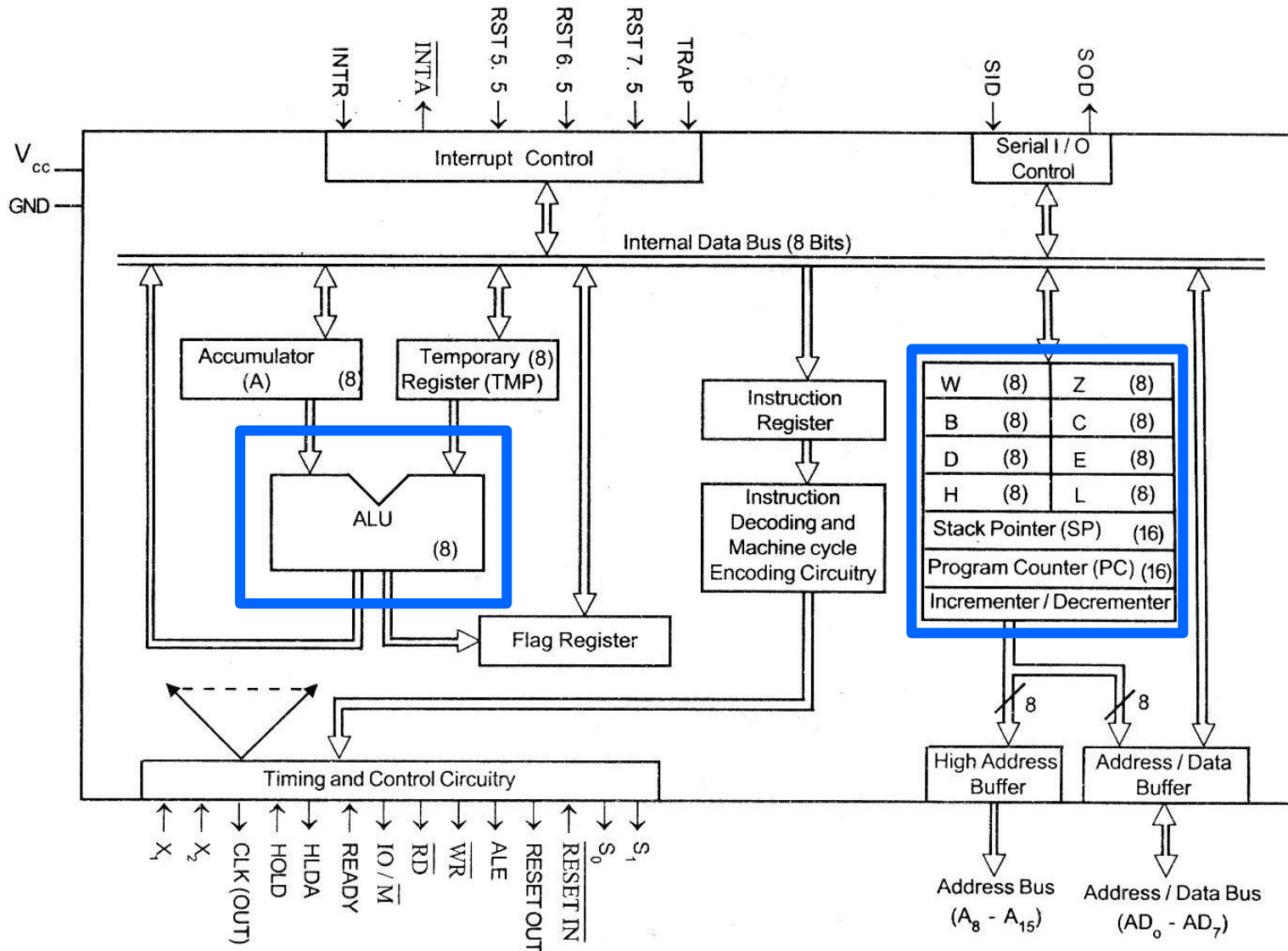
Intel 8085 (**CISC** 1976)



- The 8085 is an 8-bit processor. It can have up to 2^8 or 256 instructions
- The 8085 has a total of 80 instructions.

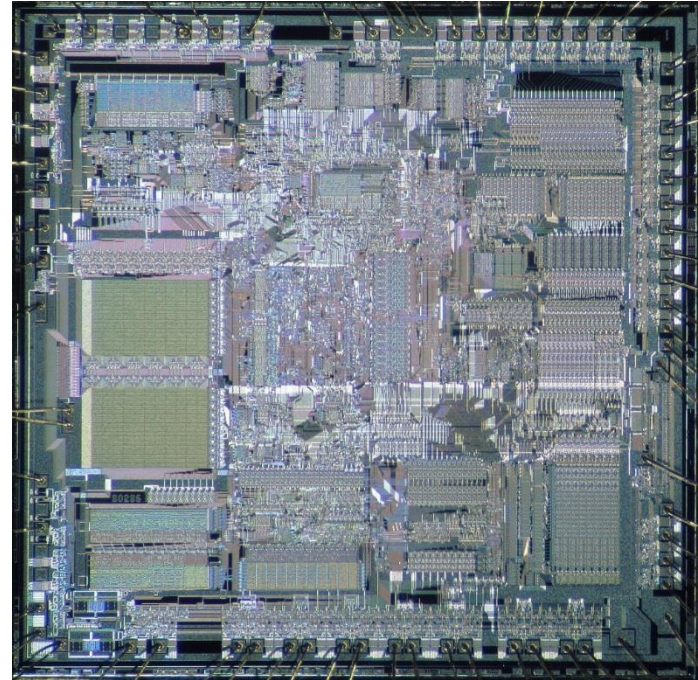
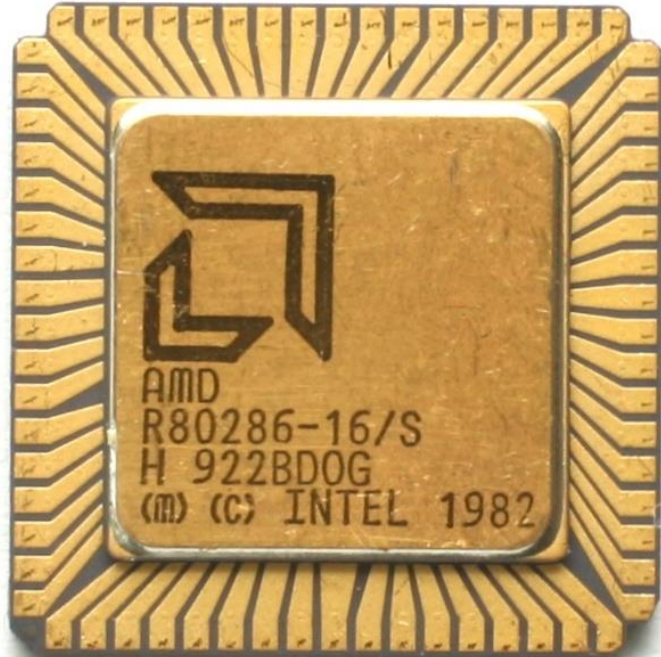


INTEL 8085 ARCHITECTURE

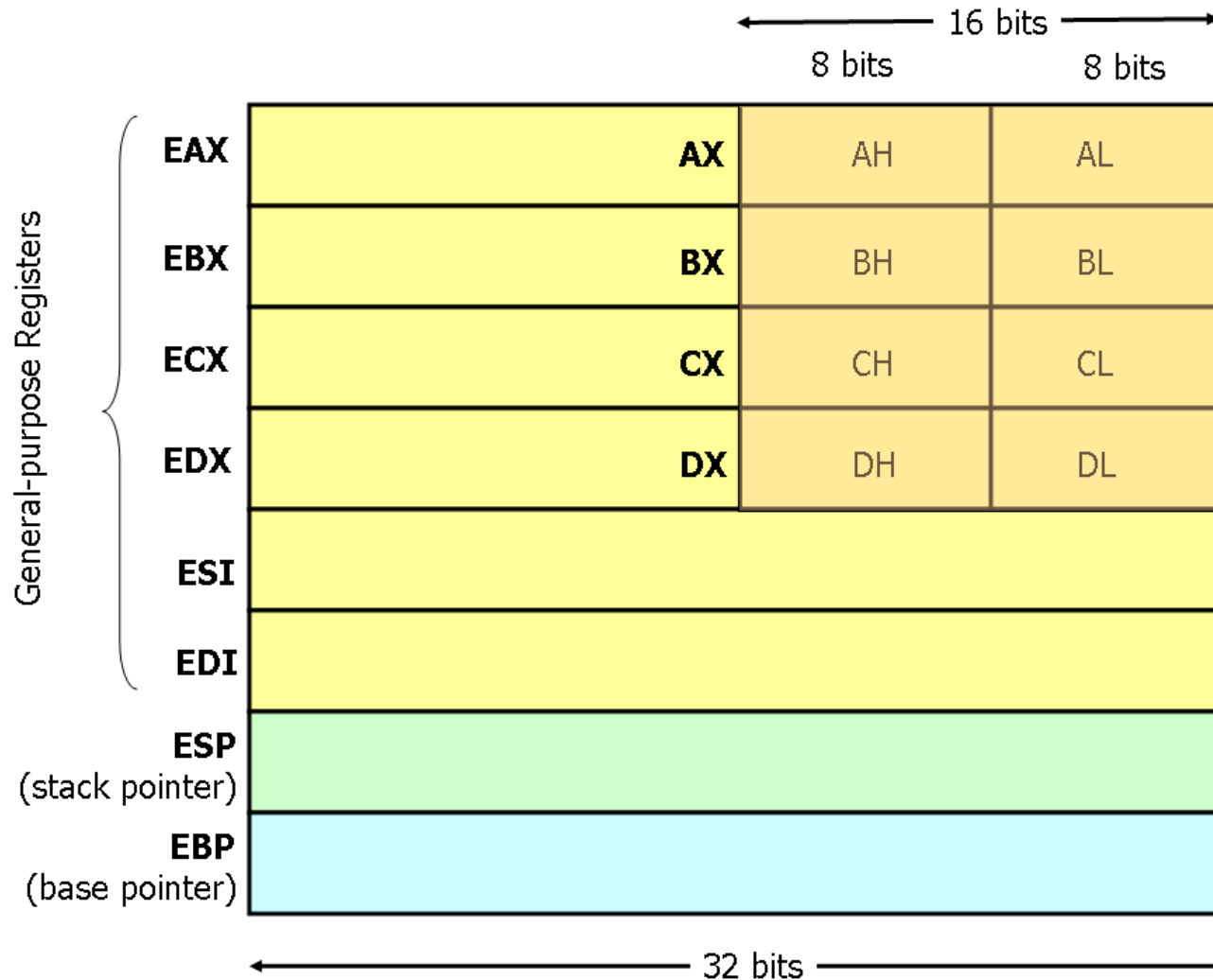


8-General Purpose Registers, (8-bit long)

INTEL 80286 (**CISC** 1980's)



INTEL 80x86 Registers



x86 Instructions: 8-16-32 bit data

Example	Meaning	Data Size
add AH, BL	$AH \leftarrow AH + BL$	8-bit
add AX, -1	$AX \leftarrow AX + 0xFFFF$	16-bit
add EAX, EDX	$EAX \leftarrow EAX + EDX$	32-bit

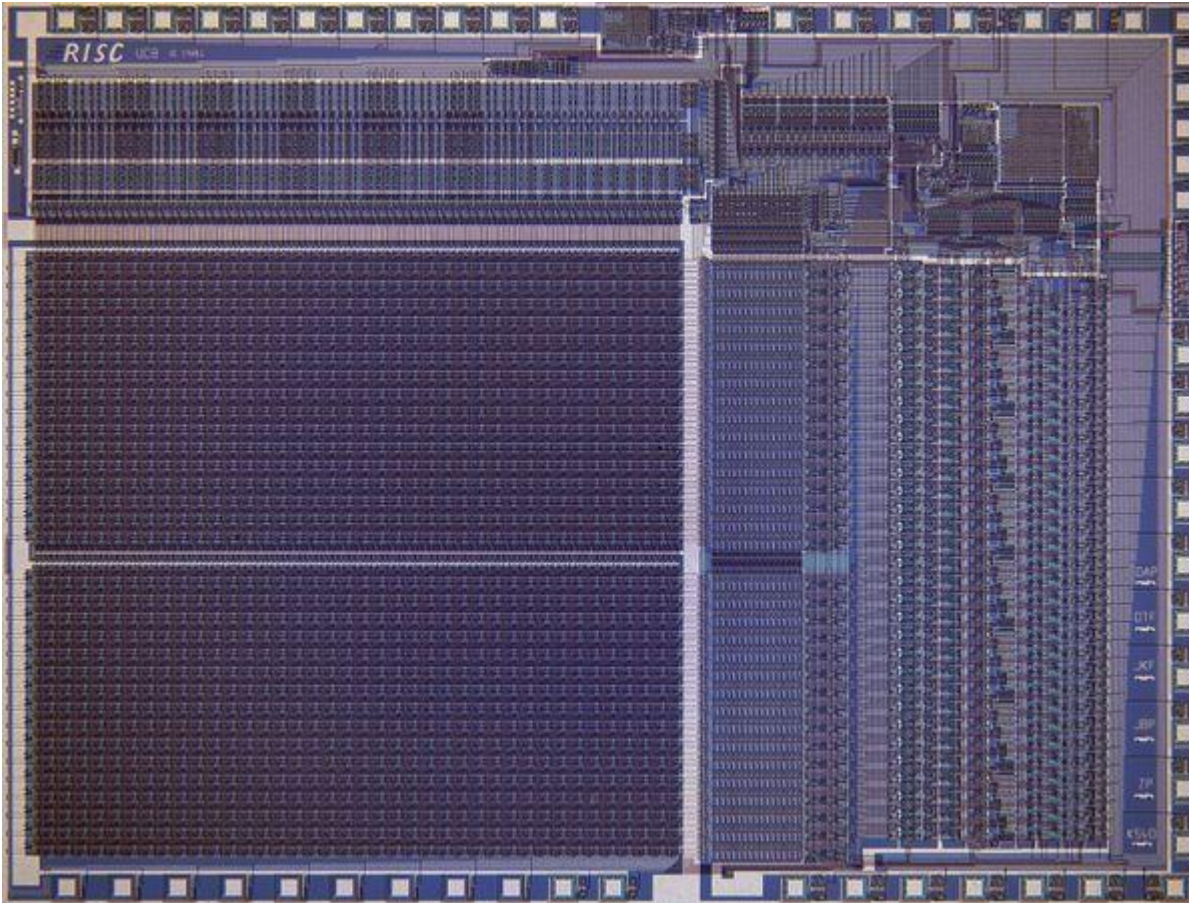


Addition using different data size

MIPS vs x86 CPU's

Feature	MIPS	x86
# of registers	32 general purpose	8, some restrictions on purpose
# of operands	3 (2 source, 1 destination)	2 (1 source, 1 source/destination)
operand location	registers or immediates	registers, immediates, or memory
operand size	32 bits	8, 16, or 32 bits
condition codes	no	yes
instruction types	simple	simple and complicated
instruction encoding	fixed, 4 bytes	variable, 1–15 bytes

RISC Processor (1982); The start



The RISC I, U.C. Berkeley's first implementation of a RISC processor in 1982, contained 44,420 transistors made with a 5 micron NMOS process. The 77 mm² chip ran at 1 MHz.

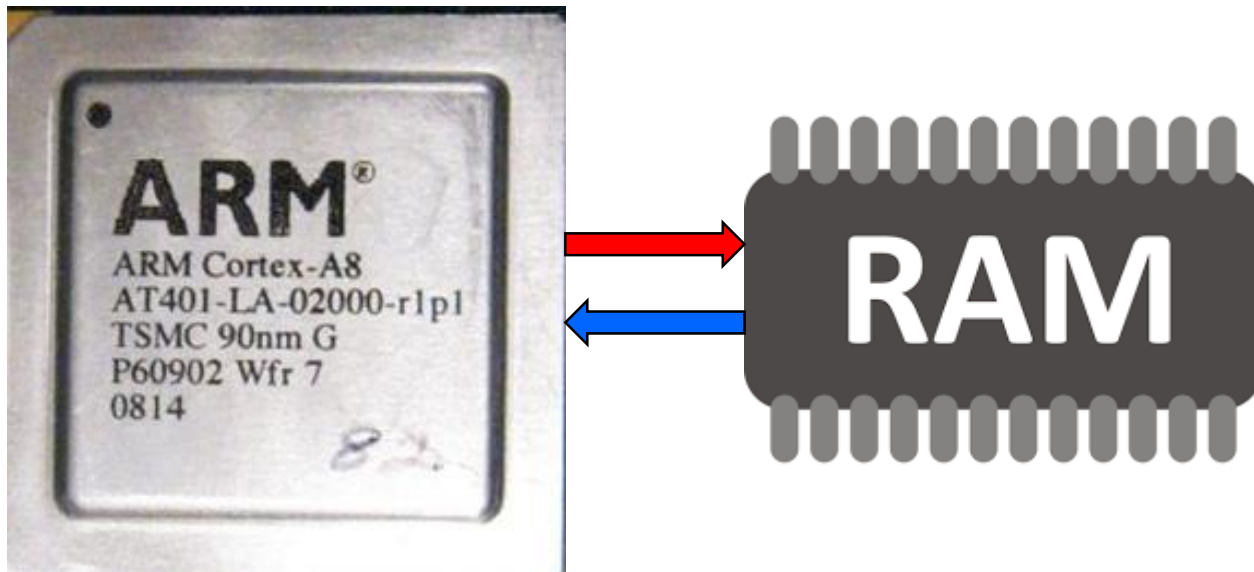
Today ... RISC-V

	RISC-V
YEAR OF BIRTH	2011
BASIC INSTRUCTION LENGTH	32
ADDRESSING	32/64
ENDIANNESS	little endian
MEMORY ACCESS	load-store
UNALIGNED LOAD/STORES	yes
BRANCH-DELAY-SLOT	no
IEEE 754 COMPLIANT	yes
REGISTERS	32 integer/ 32 floating point
OPEN SOURCE LICENSE	yes
TOTAL INSTRUCTIONS	76 (IMAF)



MIPS is a [**Load/Store** Architecture]

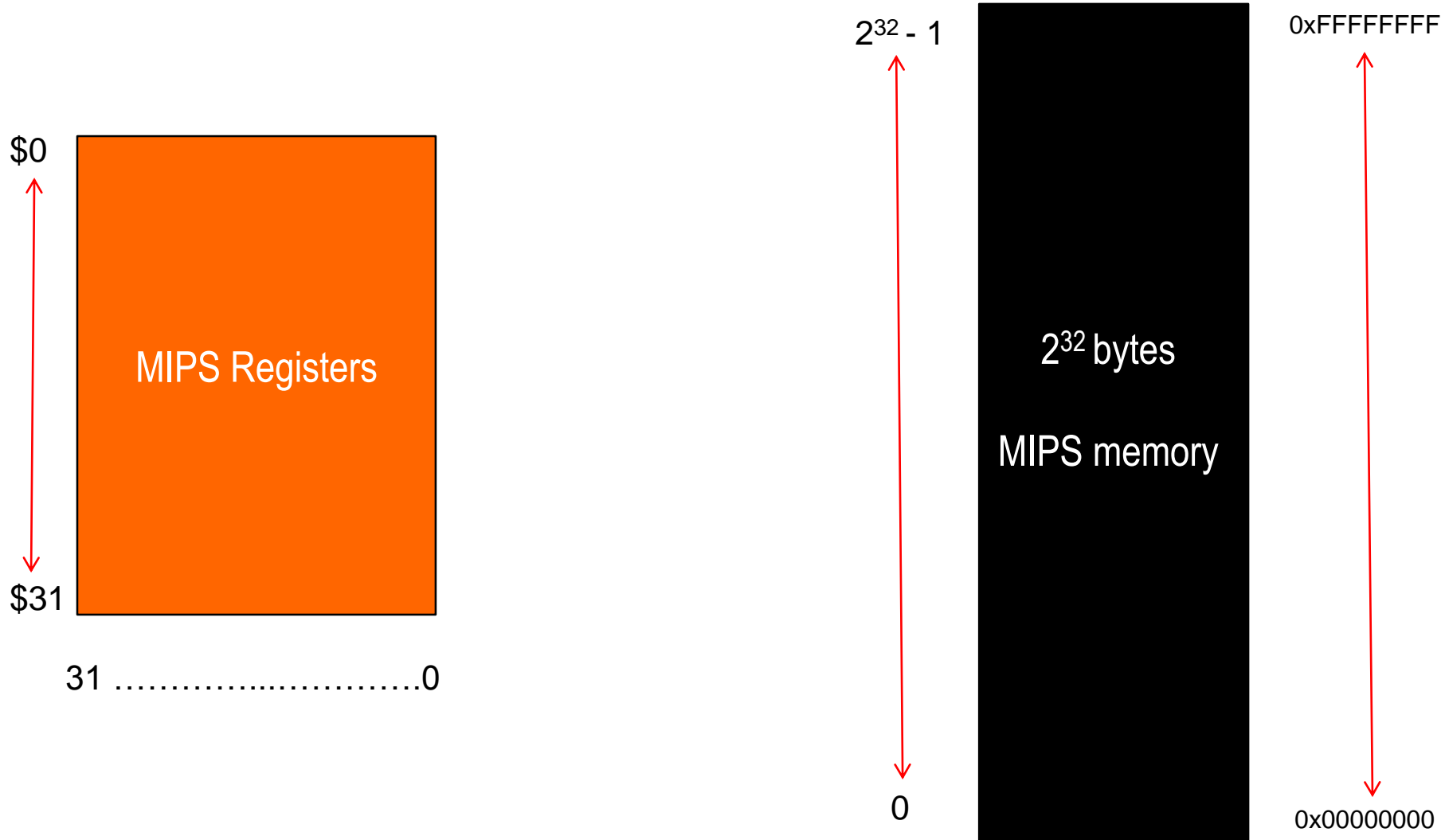
- **Load** [from memory (RAM) to CPU registers]
- **Store** [from CPU registers to memory (RAM)]



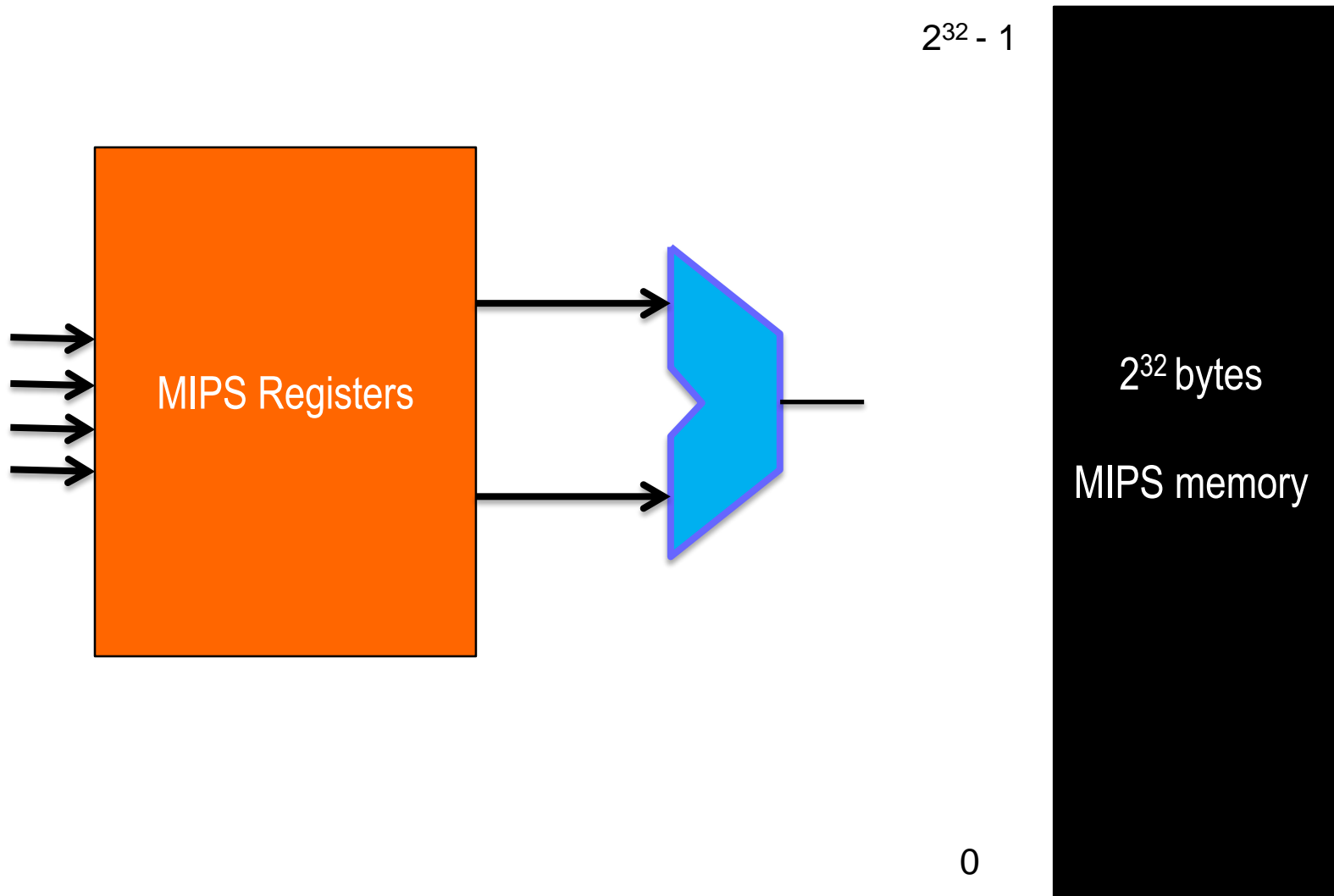
Registers ... Data transfers

- In a CPU, registers store temporary data.
 - **Fast** Operations/Transfers:
 - Between registers (**add, sub, ...**)
 - **Relatively slow** Operations/Transfers:
 - To-and-from memory (**load, store**).

MIPS: Registers and Memory



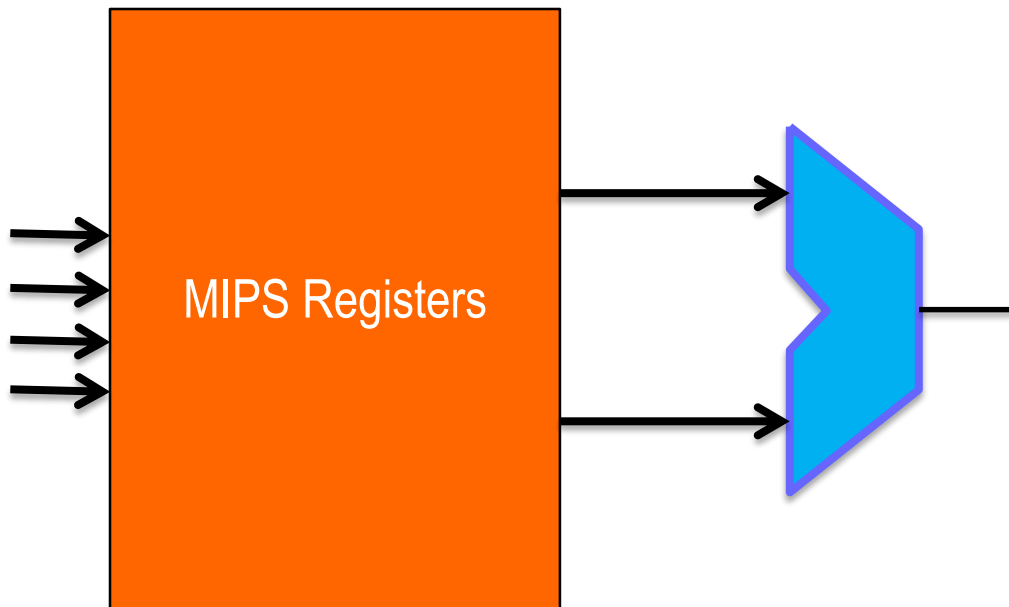
MIPS: Registers, ALU, Memory



MIPS **add** instruction

```
add $t2, $t0, $t1
```

$2^{32} - 1$

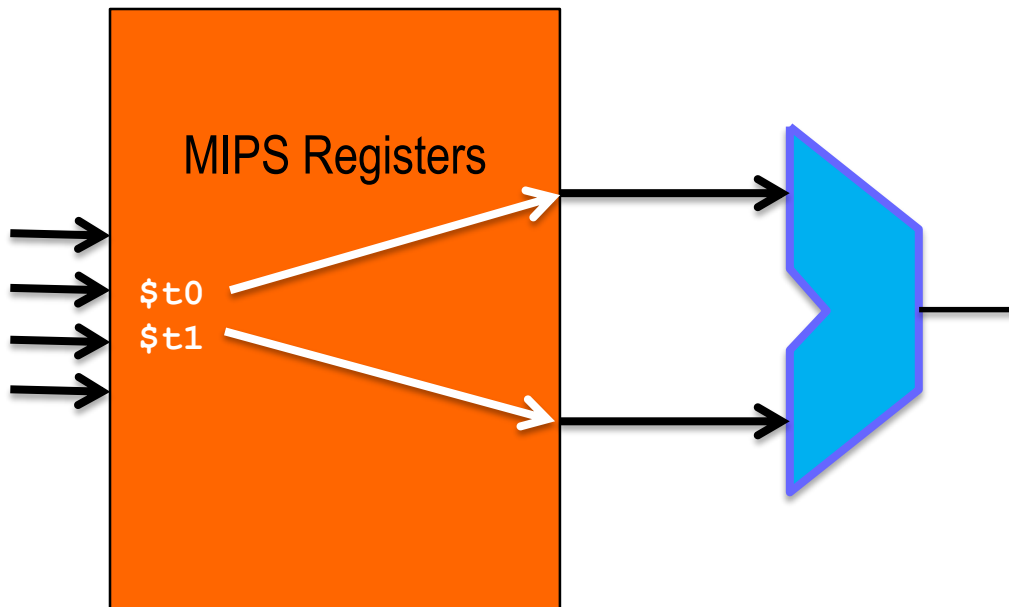


2^{32} bytes
MIPS memory

0

MIPS Instruction

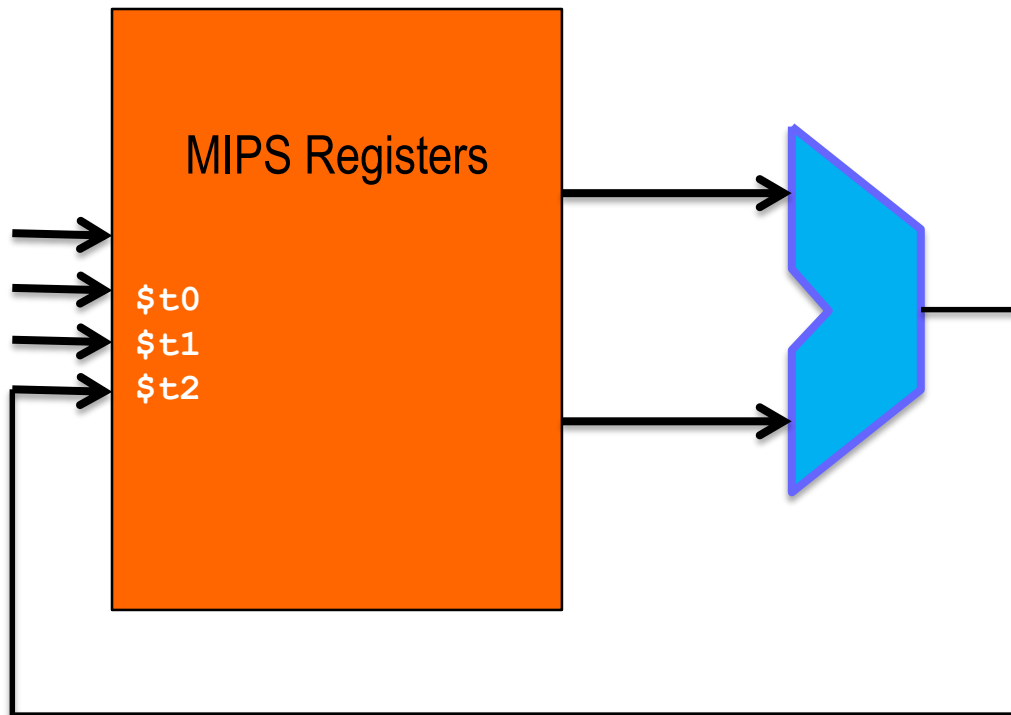
```
add $t2, $t0, $t1
```



Registers: `$t2`, `$t0`, `$t1`

MIPS Instruction

```
add $t2, $t0, $t1
```



Registers: `$t2`, `$t0`, `$t1`

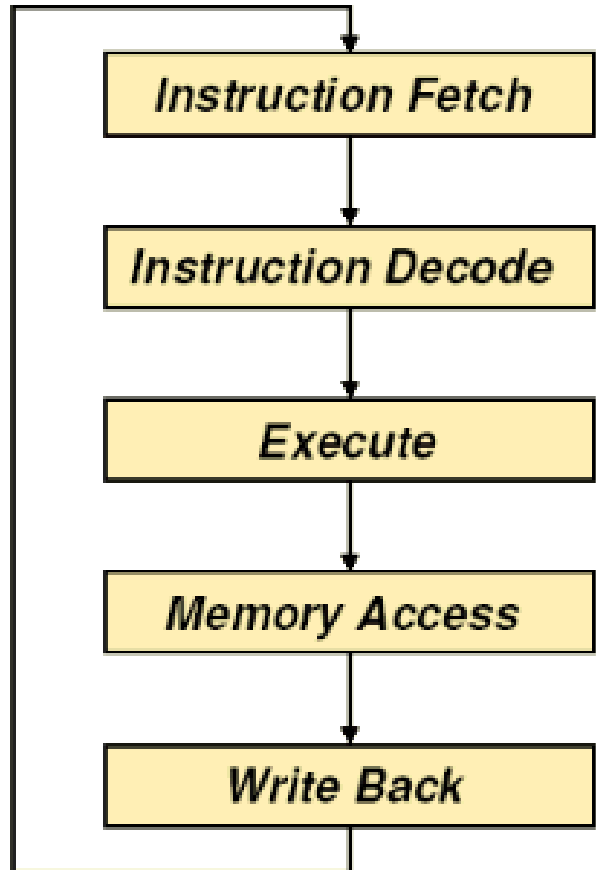
MIPS32 Register Set

Name	Register Number	Usage
<code>\$0</code>	0	the constant value 0
<code>\$at</code>	1	assembler temporary
<code>\$v0-\$v1</code>	2-3	Function return values
<code>\$a0-\$a3</code>	4-7	Function arguments
<code>\$t0-\$t7</code>	8-15	temporaries
<code>\$s0-\$s7</code>	16-23	saved variables
<code>\$t8-\$t9</code>	24-25	more temporaries
<code>\$k0-\$k1</code>	26-27	OS temporaries
<code>\$gp</code>	28	global pointer
<code>\$sp</code>	29	stack pointer
<code>\$fp</code>	30	frame pointer
<code>\$ra</code>	31	Function return address

Operands: Registers

- **Registers:**
 - \$ before name
- **Registers used for specific purposes:**
 - \$0 always holds the constant value zero: \$0
 - the *saved registers*, \$s0–\$s7, used to hold variables
 - the *temporary registers*, \$t0–\$t9, used to hold intermediate values during a larger computation
 -

Fetch-Decode-Execute-Memory-WriteBack



For each instruction

The explanation of the 5 cycle

Fetch-Decode-Execute-Memory-WriteBack

- **IF: Instruction Fetch**
 - The next instruction is read from the program memory
- **ID: Instruction Decode**
 - The instruction operand is decoded, and the registers needed are read.
- **EX: Execute**
 - Executes the instruction and performs calculations with the ALU (arithmetic and logic unit)
- **MEM: Memory Access**
 - Reads or writes to the data memory, such as with the **lw** (Load Word) and **sw** (Store Word) instructions
- **WB: Write Back**
 - Write back the results in the resulting register

Example-1

Fetch-Decode-Execute-Memory-WriteBack

```
add $t0, $s1, $s2
```

op code	rs (\$s1)	rt (\$s2)	rd (\$t0)	shamt	function
000000	10001	10010	1000	00000	100000
6-bits	5-bits	5-bits	5-bits	5-bits	6-bits

0x02324020

IF: get the (above) instruction from memory

ID: What is the instruction?.

- 000000 and 100000 is the **add**
- **Read the registers**
 - (Example: [\$s1] = 2 and [\$s2] = 3)

EX: **add** ([\$s1] + [\$s2]) = 2 + 3 = 5

MEM: No Operation

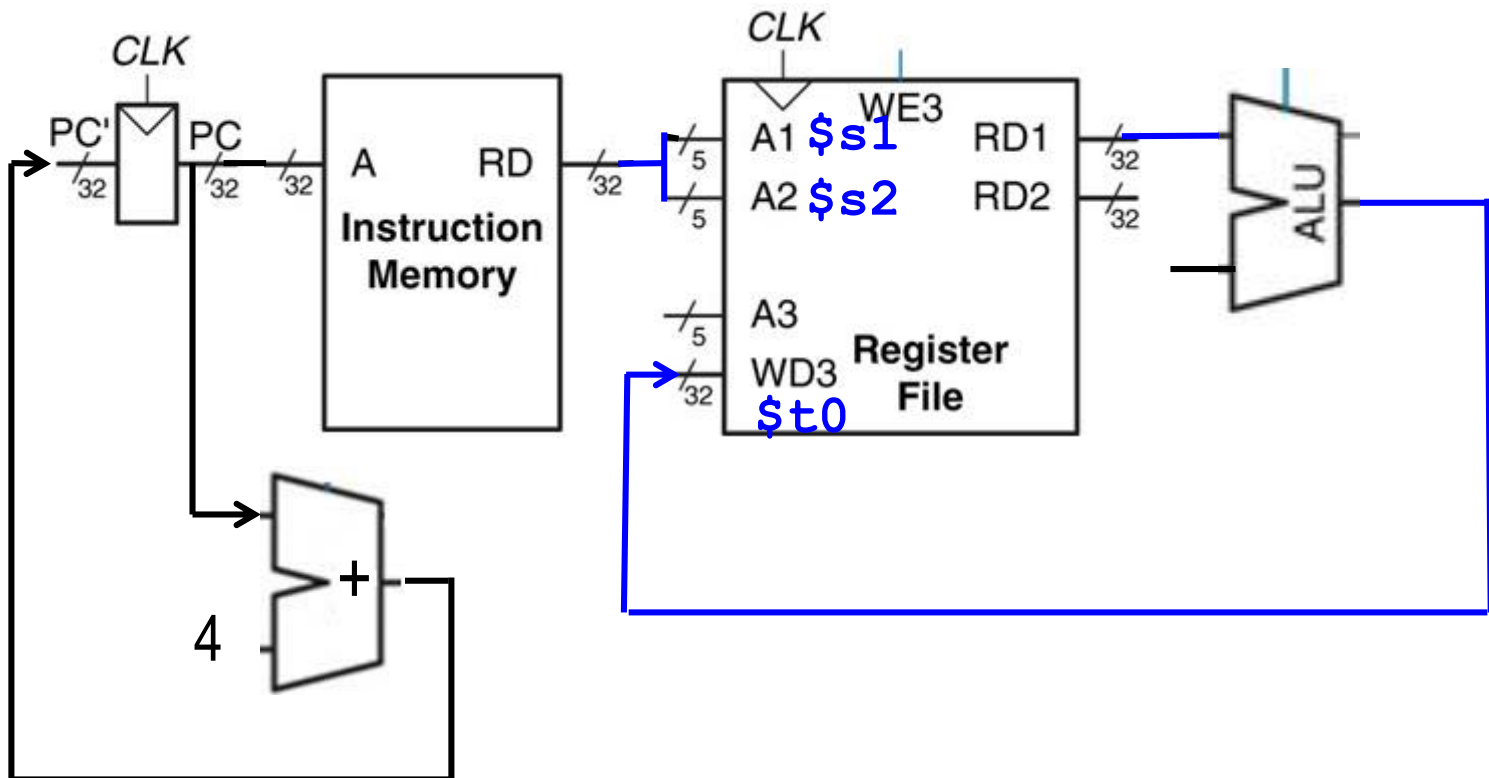
WB: The result back to reg. \$t0 = 5

add \$t0, \$s1, \$s2

Explanations

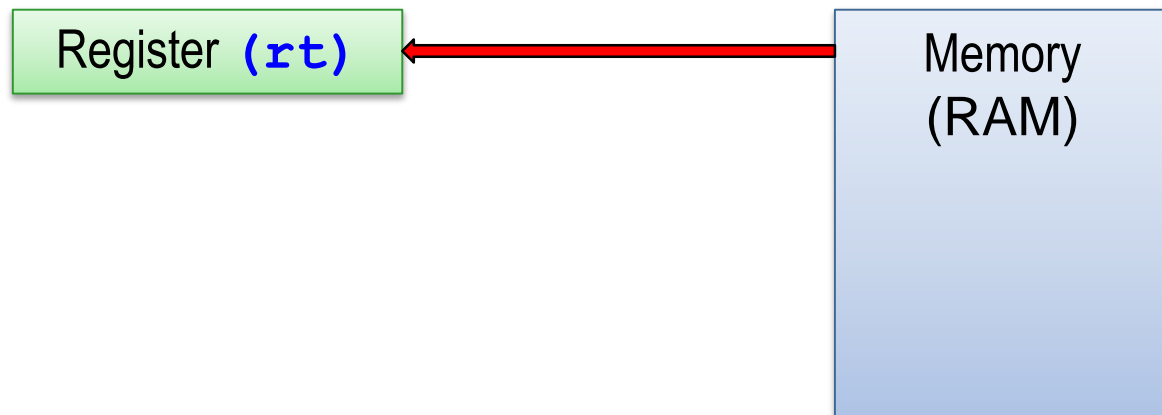
- **op** (**o**peration **c**ode)
- **rs** (First **s**ource **r**egister operand)
- **rt** (Second source **r**egister operand)
- **rd** (**D**estination **r**egister operand)
- **shamt** (**S**hift **a**mount - used in shift instructions)
- **funct** (Select the variant of the operation in the op code field)

add \$t0, \$s1, \$s2



lw \$rt, offset(\$rs)

Read (Copy) a word from memory [**offset(\$rs)**] into register (**rt**)



Indexed (Based) Addressing

Arithmetic expressions and MIPS instructions

EXAMPLES

MIPS32: Example-1;

Expression: `a = b + c;`

- What is the corresponding MIPS code?

```
$t0 = a  
$s1 = b  
$s2 = c
```

```
add $tx, $sx, $sx
```

MIPS32: Example-1; Solution

Expression: `a = b + c;`

- What is the corresponding MIPS code?

MIPS Code

```
add $t0, $s1, $s2
```

```
$t0 = a  
$s1 = b  
$s2 = c
```

Example-2

MIPS32: Example-2;

Expressions: $a = b + c + d;$

What is the corresponding MIPS code?

```
$s0 = a  
$s1 = c  
$s2 = d  
$s3 = b
```

```
add $tx, $sx, $sx  
add $sx, $tx, $sx
```


MIPS32: Example-2; Solution

Expressions: $a = b + c + d;$

MIPS Code

```
add $t0, $s1, $s2 > c + d
add $s0, $t0, $s3 > b + (c + d)
```

```
$s0 = a
$s1 = c
$s2 = d
$s3 = b
```

Example-3

New instruction (**sub**)

sub \$t0, \$s1, \$s2
is equivalent to:
\$t0 = \$s1 - s2

MIPS32: Example-3;

Expressions: $a = b + c + d;$
 $e = f - a;$

What is the corresponding MIPS code?

$\$s0 = a$
 $\$s1 = c$
 $\$s2 = d$
 $\$s3 = b$
 $\$s4 = e$
 $\$s5 = f$

$\text{sub } \$t0, \$s1, \$s2$
is equivalent to:
 $\$t0 = \$s1 - s2$

$\text{add } \$tx, \$sx, \$sx$
 $\text{add } \$sx, \$tx, \$sx$
 $\text{sub } \$sx, \$sx, \$sx$

MIPS32: Example-3; Solution

Expressions: $a = b + c + d;$
 $e = f - a;$

MIPS Code

```
add $t0, $s1, $s2 > c + d
add $s0, $t0, $s3 > b + (c + d)
sub $s4, $s5, $s0 > f - [ b + (c + d) ]
```

```
$s0 = a
$s1 = c
$s2 = d
$s3 = b
$s4 = e
$s5 = f
```

Example-4

MIPS32: Example-4;

Expression: $a = (b + c) - (d + e);$

What is the corresponding MIPS code?

$\$s0$	=	a
$\$s1$	=	b
$\$s2$	=	c
$\$s3$	=	d
$\$s4$	=	e

```
add $tx, $sx, $sx  
sub $sx, $tx, $tx
```

MIPS32: Example-4; Solution

Expression: $a = (b + c) - (d + e) ;$

MIPS Code

```
add $t0, $s1, $s2 > (b+c)
add $t1, $s3, $s4 > (d+e)
sub $s0, $t0, $t1 > a = (b+c) - (d+e)
```

```
$s0 = a
$s1 = b
$s2 = c
$s3 = d
$s4 = e
```


Example-5

MIPS32: Example-5

Expression: $a = b + c + d - e$

What is the corresponding MIPS code?

Choose the registers

MIPS32: Example-5; Solution

Expression: $a = b + c + d - e$

MIPS Code

```
add $t0, $s1, $s2    >  c + d
add $s4, $t0, $s3    >  b + (c+d)
sub $s0, $t0, $s4    >  [b + (c+d)] - e
```

Another MIPS instruction

addi

addi ... **add-i**mmEDIATE

```
a = b + 10; (10 is decimal)
```

MIPS Code

```
addi $s0,$s1,10
```

```
a = b - 10; (-10 is decimal)
```

MIPS Code

```
addi $s0,$s1,-10
```


MARS Software

Download MARS

MARS software

- Before the next class please download the MIPS-MARS software
- It is just one small file ... [for PC's and MAC's]

Software <http://courses.missouristate.edu/kenvollmar/mars/>

A green, rounded rectangular button with the text "Download MARS" in a dark, sans-serif font.