UNIT 1: THE PROCESSOR

Estructura de Computadores (Computer Organization)

Course 2018/2019

ETS Ingeniería Informática

Universitat Politècnica de València

Unit goals

- To understand the resources needed for the execution of a simple, yet sufficient instruction subset of MIPS
- To understand the phases involved in the execution of instructions
- To design a simple datapath and hardwired control unit for a limited subset of MIPS
- To evaluate the design and to explore further alternatives

- Context: MIPS 32 architecture, using a reduced version of the MIPS R2000 processor

Unit contents

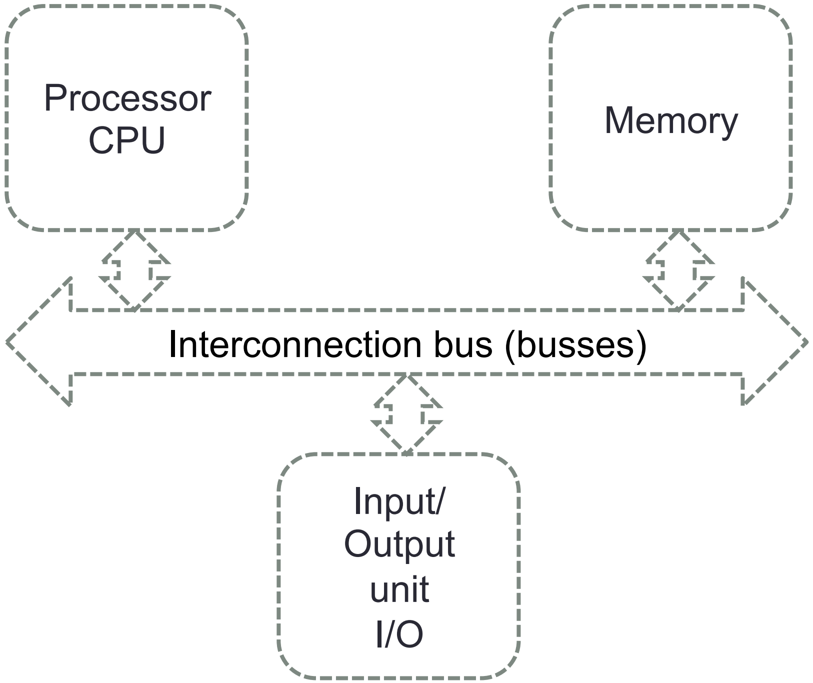
- 1 An overview of the MIPS32 architecture
- 2 Datapath design
  - Introduction
  - Instruction subset
  - Components
  - Execution phases
  - Complete datapath
    - R-type instructions
    - Memory access instructions
    - Conditional branch instruction (beq)
      - Unconditional branch instructions (j, jr) in lab session 10
- 3 Control unit design
  - Hardwired CU
  - Evaluation
  - Other alternatives

Bibliography

- D. Patterson, J. Hennessy. *Computer organization and design. The hardware/software interface*. 4th edition. 2009. Elsevier
  - Chapter 2 and chapter 4, sections 4.1 to 4.4
  - Annex D for further reading on microprogrammed control unit

1. MIPS32 architecture: an overview

Components of a computer system



Architecture vs. implementation

- Architecture
  - Denotes the instruction set, registers, memory management (including virtual memory), memory map, exception system, etc.
    - In summary, anything the *system programmer* needs to know about the underlying machinery
- Implementation
  - The realization of the features specified by the architecture – or how the processor applies the architecture specification

Instruction set version (*)	Registers width	Processors
MIPS I	32	R2000, R3000
MIPS II	32	R6000
MIPS III	64	R4000
MIPS IV	64	R5000, R10000,…
MIPS V	64	-

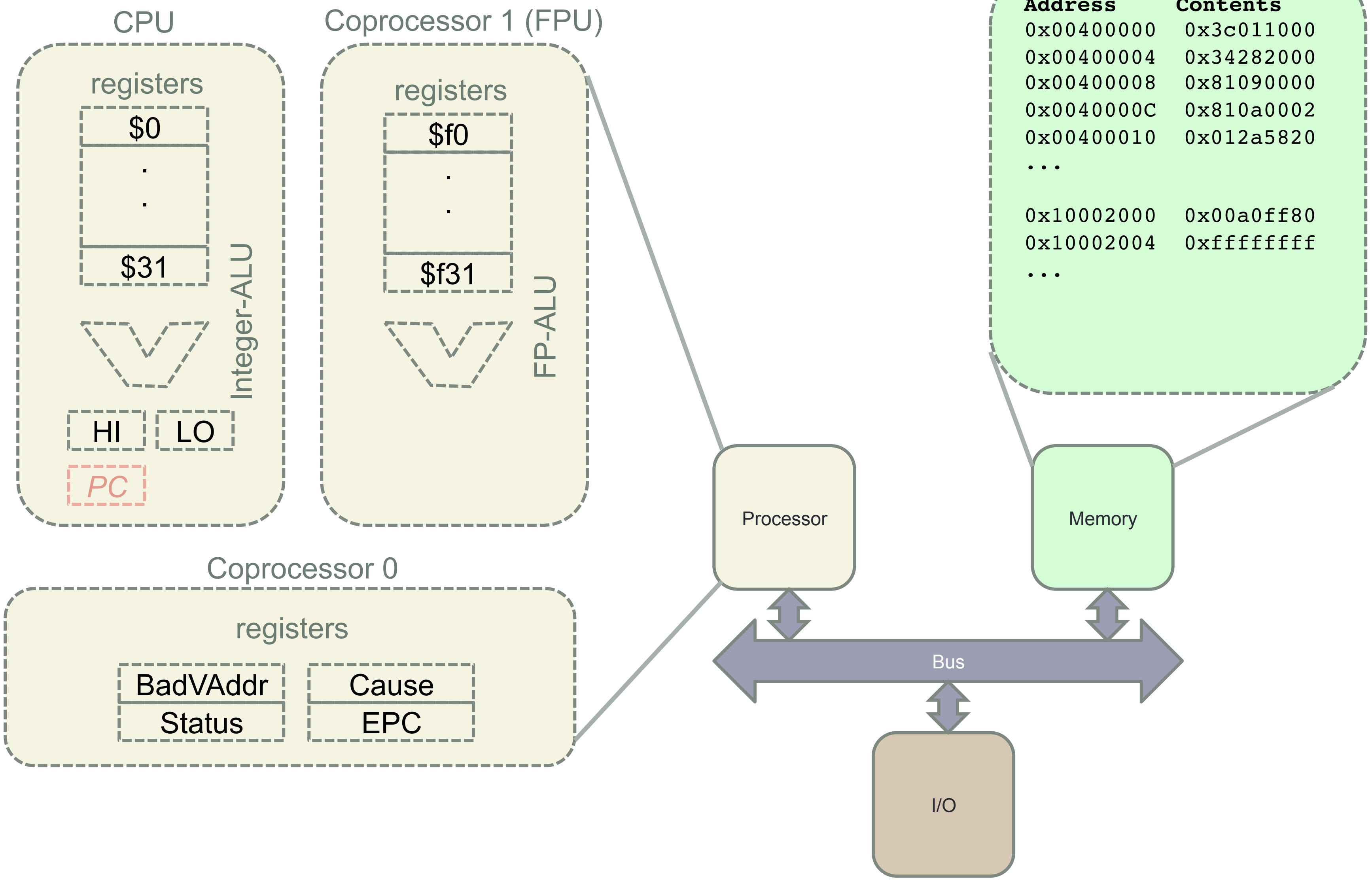
(*) Note: MIPS32 is a superset of MIPS I and MIPS II

Further info:
https://en.wikipedia.org/wiki/List_of_MIPS_microarchitectures
https://imgtec.com/mips/architectures

Main features of MIPS32 architecture

- RISC architecture (*Reduced Instruction Set Computer*)
  - A processor design especially amenable to *pipelining* (Unit 2)
  - MIPS = **M**icroprocessor with **I**nterlocked **P**ipeline **S**tages)
- 32-bit word size (largest data handled by single instruction)
- Main data sizes
  - byte (8-bit, *B*), half word (16-bit, *H*), word (32-bit, *W*)
- Load/store architecture
  - Only *load* and *store* instructions do access data in memory
  - All operands for arithmetic/logic instructions reside in registers
  - 3-operand A/L instructions (32-bit registers)
- Main operating modes
  - User – restricted, for regular user programs
  - Kernel – unrestricted, for the OS

Hardware model



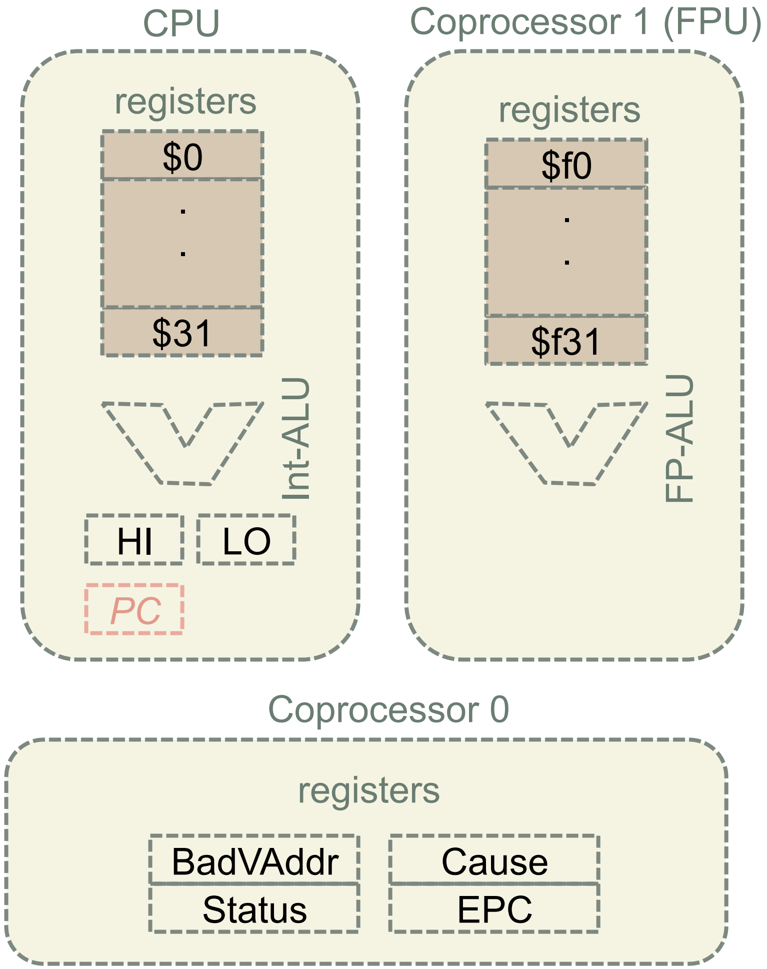
Address	Contents
0x00400000	0x3c011000
0x00400004	0x34282000
0x00400008	0x81090000
0x0040000C	0x810a0002
0x00400010	0x012a5820
...	
0x10002000	0x00a0ff80
0x10002004	0xffffffff
...	

Programming model – resources

- CPU
  - Thirty two 32-bit registers ($0..$31)
  - Integer Arithmetic-Logic Unit
  - HI and LO registers used for integer multiplication and division
  - Program counter (PC) points to next instruction to execute. The PC is not architecturally-visible, but is indirectly modified by certain instructions
- Coprocessor 0
  - Control resources (OS calls, operating modes, interrupts…)
    - 4 registers shown in previous slide, but there are more
- Coprocessor 1 (floating-point unit, optional)
  - Thirty two 32-bit floating-point registers ($f0..$f31)
  - Floating-Point Unit
    - Single-precision arithmetic (32 registers)
    - Double-precision arithmetic (16 pairs of 32-bit registers)

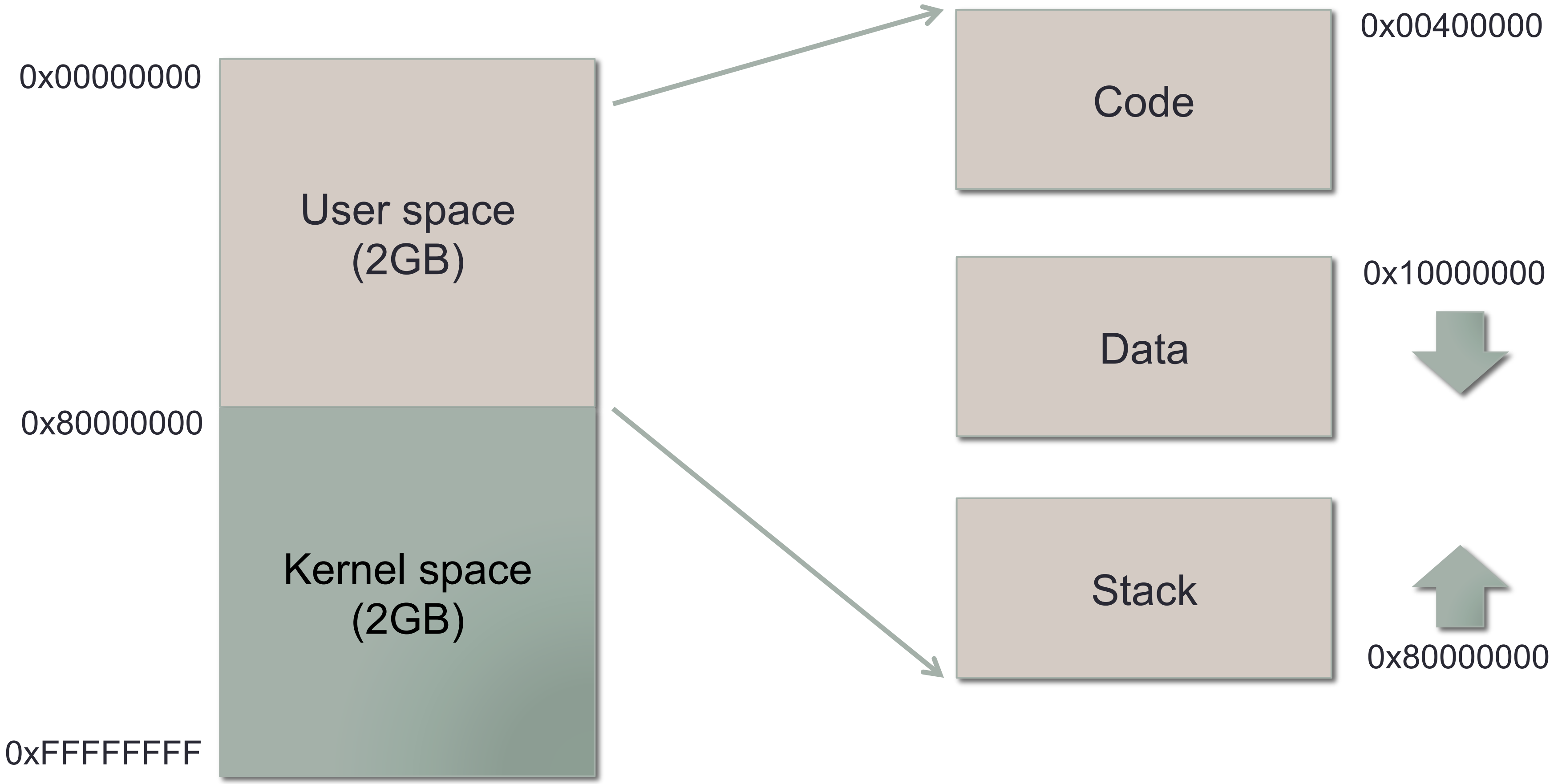
Programming model – registers

Name	Reg Nr.	Conventional use – not enforced by the hardware, except (*)
$zero	$0	Hardwired, constant 0 (*)
$at	$1	Assembler temporary
$v0-$v1	$2-$3	Return of function results
$a0-$a3	$4-$7	Function arguments (parameters)
$t0-$t7	$8-$15	Temporary registers
$s0-$s7	$16-$23	Callee-saved temporary registers
$t8-$t9	$24-$25	Temporary registers
$k0-$k1	$26-$27	Used by the OS
$gp	$28	Global pointer (static global vars.)
$sp	$29	Stack pointer (local vars.)
$fp	$30	Frame pointer
$ra	$31	Return address (*)
$f0-$f31	$f0..$f31	Floating-point registers



Programming model – the memory

- Addressing space



Programming model – the memory

- The memory layout of a program can be determined with assembly directives
  - Memory segment directives
    - `.data` [*address*] – start of a data segment
    - `.text` [*address*] – start of a code segment
    - `.end` – end of program code
  - Data allocation directives
    - `.byte` *b1 [,b2] …* – bytes with initial values
    - `.half` *h1 [,h2] …* – half words (16-bit)
    - `.word` *w1 [,w2] …* – words (32-bit)
    - `.space` *n* – reserves *n* bytes
    - `.ascii` *string1 [,string2] …* – char strings
    - `.asciiz` *string1 [,string2] …* – null-ended char strings
  - *Labels (`A`, `W`, `V`, `C1`, `C2`) improve readability – and make life easier!*

```
          .data 0x10000000
A:        .byte 2, 3, 4
W:        .word 33
V:        .space 100

          .data 0x10004000
C1:       .ascii “hello”
C2:       .asciiz “goodbye”

          .text 0x00400000
          lb $t0,A
          lw $t1,W
          ...
          .end
```

Programming model – the memory

- Addressing
  - Byte-level addressing – every byte in memory has its own address
    - A byte is the smallest amount of accessible data in memory
  - MIPS32 supports both big-endian and little-endian modes
- Data alignment
  - A byte may reside in any address
  - A half word must start in an even address (multiple of 2)
  - A word must start in an address multiple of 4
    - (In MIPS64, *double words* correspondingly start in multiples of 8)
  - The assembler directive

    `.align` *N*

    overrides automatic alignment of .half, .word, etc. and enforces alignment to next address multiple of 2^N, creating a “gap” in the data segment, if needed

    `.align 0` switches auto-alignment off – packed data

Programming model – instructions

- Instruction groups

  Arithmetic

  Shift and rotate

  Register transfer

  Load and store

  Others

  Logical (bitwise)

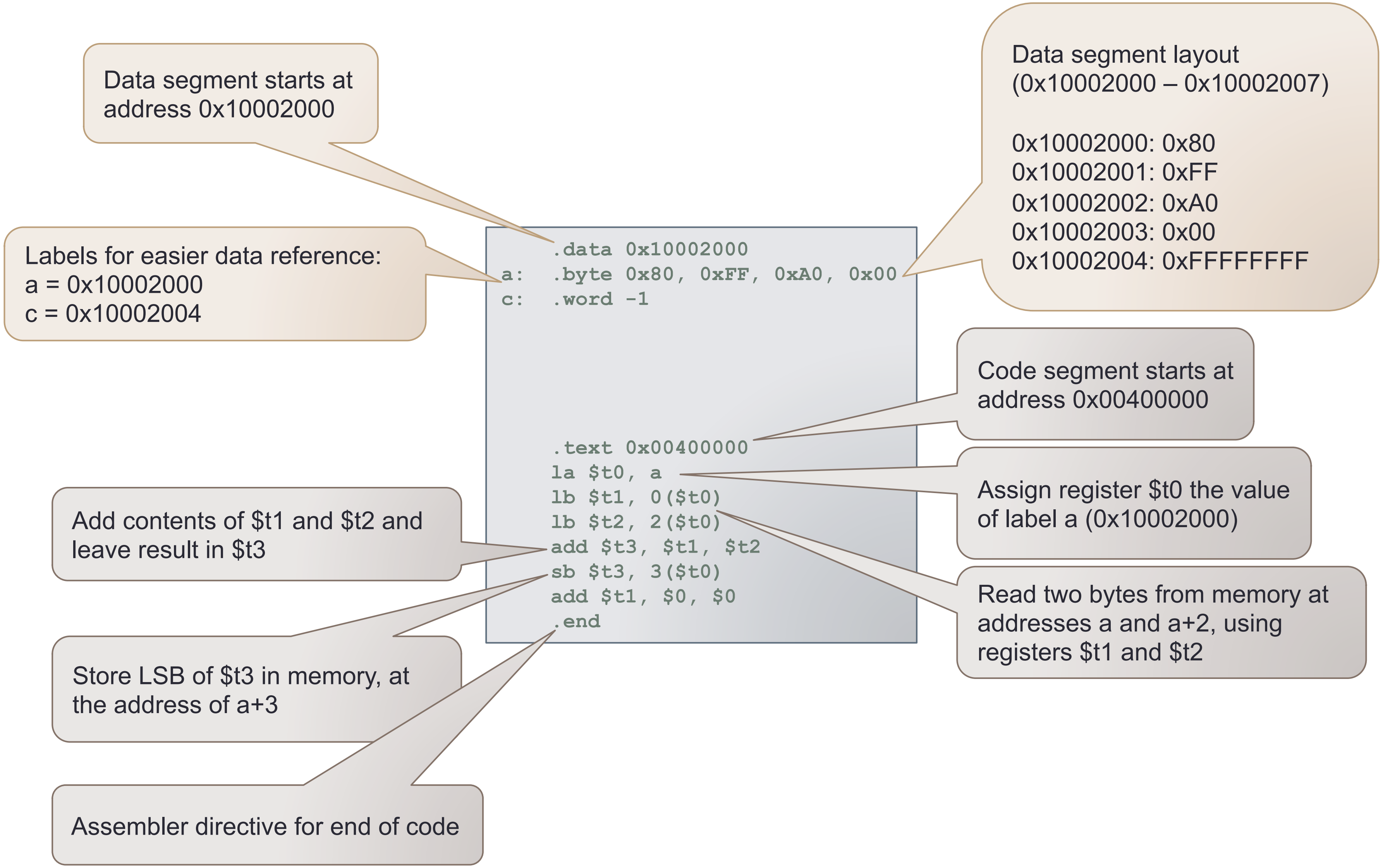
  Condition test (comparison)

  Floating-point

  Branch

- Syntax and encoding
  - *http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html*
  - *http://en.wikipedia.org/wiki/MIPS_architecture*
- Instructions vs. pseudo-instructions
  - Instructions are directly supported by the hardware
  - Pseudo-instructions require the assembler to generate proper instructions

Code in action: example



Data segment starts at address 0x10002000

Labels for easier data reference:
a = 0x10002000
c = 0x10002004

```
      .data 0x10002000
a:    .byte 0x80, 0xFF, 0xA0, 0x00
c:    .word -1




      .text 0x00400000
      la $t0, a
      lb $t1, 0($t0)
      lb $t2, 2($t0)
      add $t3, $t1, $t2
      sb $t3, 3($t0)
      add $t1, $0, $0
      .end
```

Data segment layout
(0x10002000 – 0x10002007)

0x10002000: 0x80
0x10002001: 0xFF
0x10002002: 0xA0
0x10002003: 0x00
0x10002004: 0xFFFFFFFF

Code segment starts at address 0x00400000

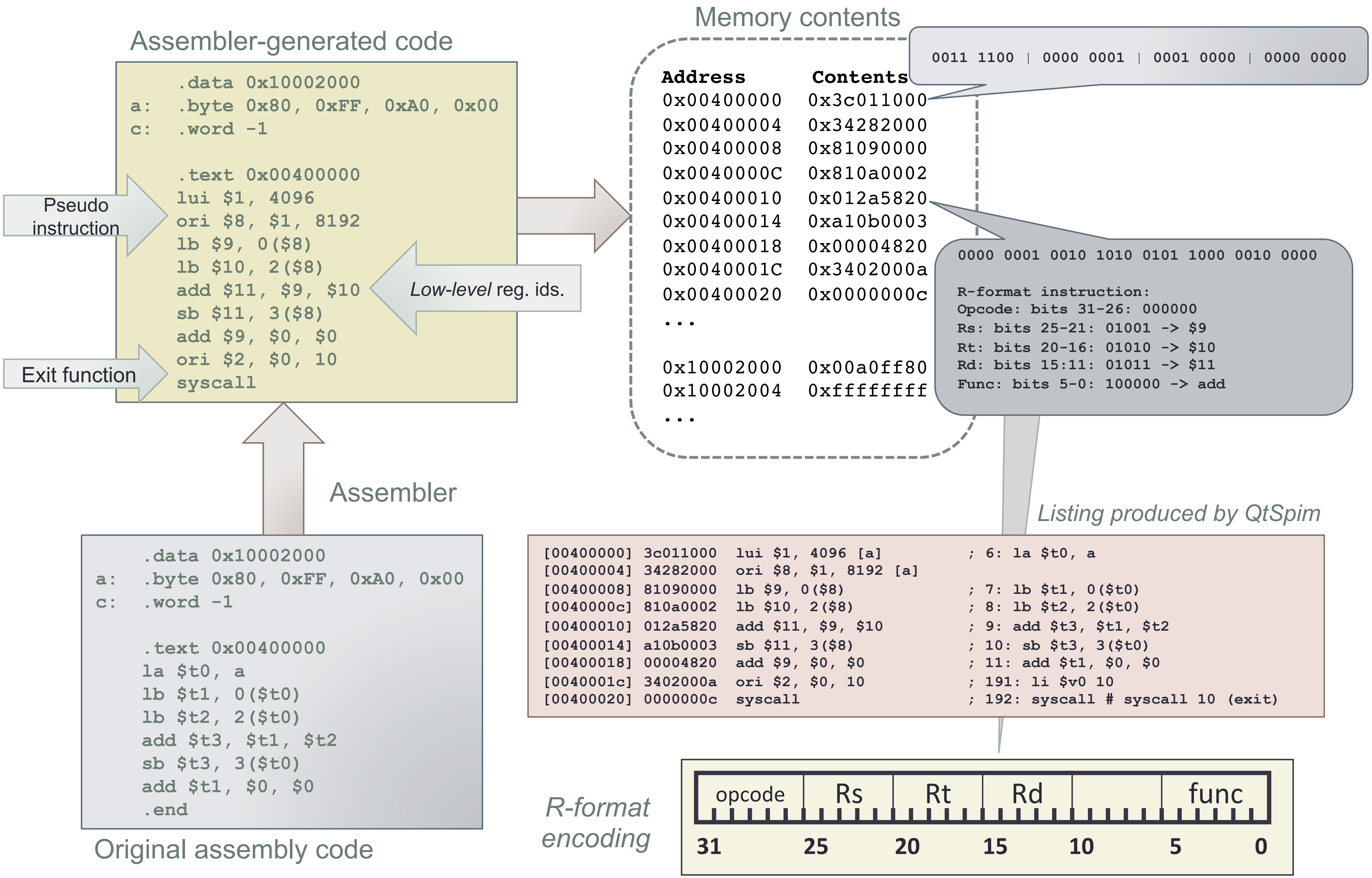
Assign register $t0 the value of label a (0x10002000)

Read two bytes from memory at addresses a and a+2, using registers $t1 and $t2

Add contents of $t1 and $t2 and leave result in $t3

Store LSB of $t3 in memory, at the address of a+3

Assembler directive for end of code



Assembler-generated code

```
    .data 0x10002000
a:  .byte 0x80, 0xFF, 0xA0, 0x00
c:  .word -1

    .text 0x00400000
    lui $1, 4096
    ori $8, $1, 8192
    lb $9, 0($8)
    lb $10, 2($8)
    add $11, $9, $10
    sb $11, 3($8)
    add $9, $0, $0
    ori $2, $0, 10
    syscall
```

Pseudo instruction

Low-level reg. ids.

Exit function

Memory contents

```
Address     Contents
0x00400000  0x3c011000
0x00400004  0x34282000
0x00400008  0x81090000
0x0040000C  0x810a0002
0x00400010  0x012a5820
0x00400014  0xa10b0003
0x00400018  0x00004820
0x0040001C  0x3402000a
0x00400020  0x0000000c
...

0x10002000  0x00a0ff80
0x10002004  0xffffffff
...
```

```
0011 1100 | 0000 0001 | 0001 0000 | 0000 0000
```

```
0000 0001 0010 1010 0101 1000 0010 0000

R-format instruction:
Opcode: bits 31-26: 000000
Rs: bits 25-21: 01001 -> $9
Rt: bits 20-16: 01010 -> $10
Rd: bits 15:11: 01011 -> $11
Func: bits 5-0: 100000 -> add
```

Assembler

```
    .data 0x10002000
a:  .byte 0x80, 0xFF, 0xA0, 0x00
c:  .word -1

    .text 0x00400000
    la $t0, a
    lb $t1, 0($t0)
    lb $t2, 2($t0)
    add $t3, $t1, $t2
    sb $t3, 3($t0)
    add $t1, $0, $0
    .end
```

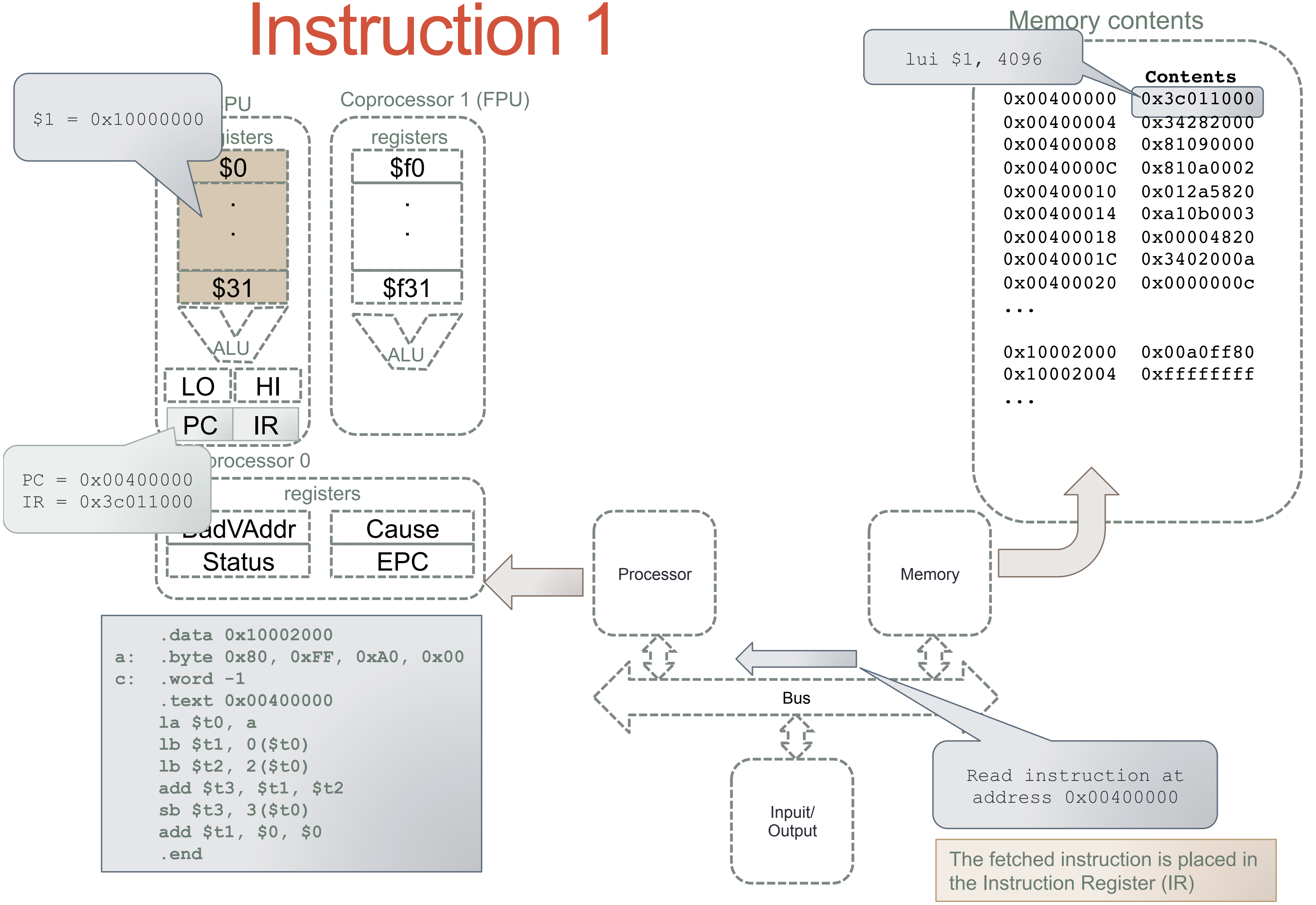
Original assembly code

Listing produced by QtSpim

```
[00400000] 3c011000  lui $1, 4096 [a]        ; 6: la $t0, a
[00400004] 34282000  ori $8, $1, 8192 [a]
[00400008] 81090000  lb $9, 0($8)            ; 7: lb $t1, 0($t0)
[0040000c] 810a0002  lb $10, 2($8)           ; 8: lb $t2, 2($t0)
[00400010] 012a5820  add $11, $9, $10        ; 9: add $t3, $t1, $t2
[00400014] a10b0003  sb $11, 3($8)           ; 10: sb $t3, 3($t0)
[00400018] 00004820  add $9, $0, $0          ; 11: add $t1, $0, $0
[0040001c] 3402000a  ori $2, $0, 10          ; 191: li $v0 10
[00400020] 0000000c  syscall                 ; 192: syscall # syscall 10 (exit)
```

R-format encoding

Instruction 1



$1 = 0x10000000

PC = 0x00400000
IR = 0x3c011000

Memory contents

lui $1, 4096

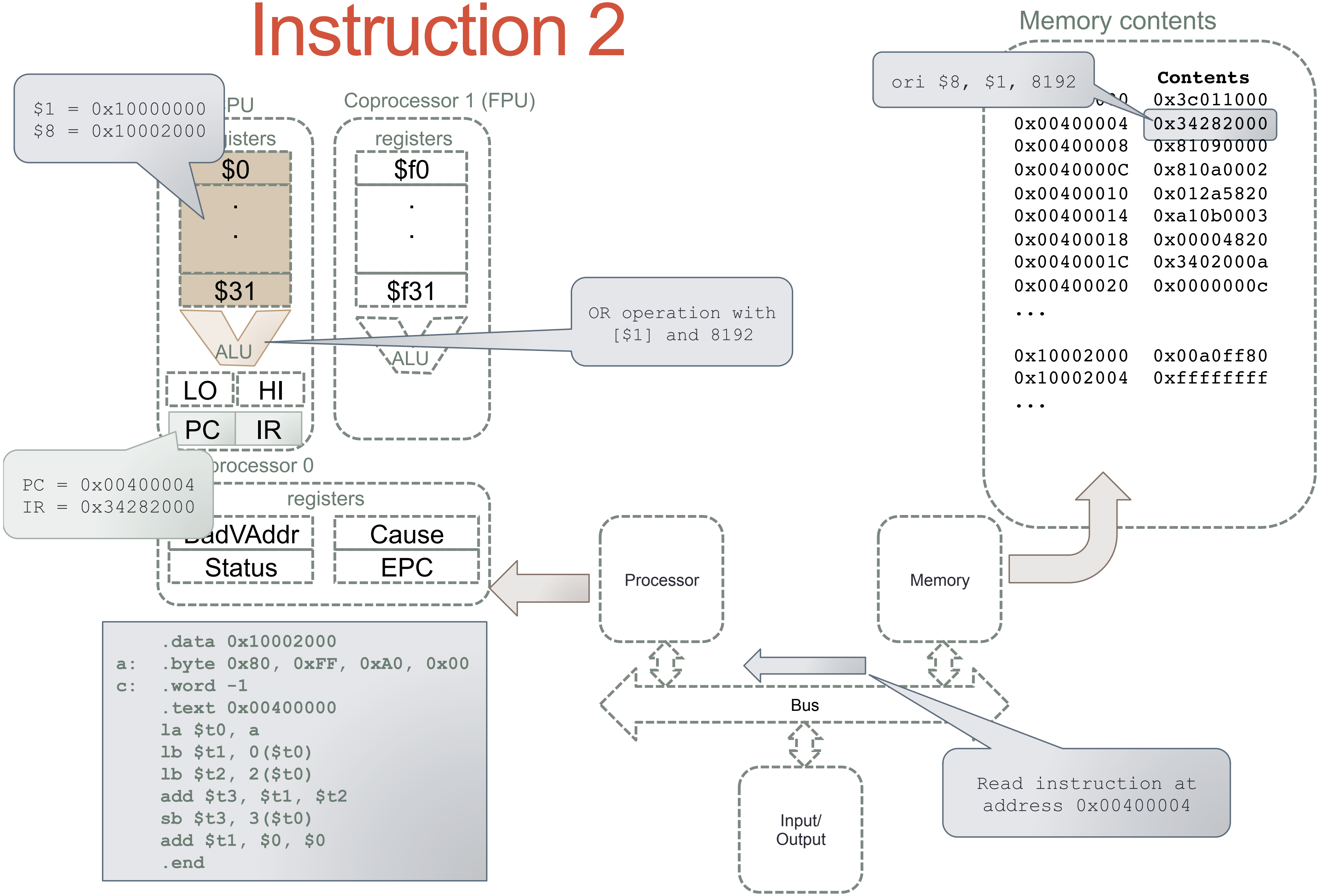
	Contents
0x00400000	0x3c011000
0x00400004	0x34282000
0x00400008	0x81090000
0x0040000C	0x810a0002
0x00400010	0x012a5820
0x00400014	0xa10b0003
0x00400018	0x00004820
0x0040001C	0x3402000a
0x00400020	0x0000000c
...	
0x10002000	0x00a0ff80
0x10002004	0xffffffff
...	

```
    .data 0x10002000
a:  .byte 0x80, 0xFF, 0xA0, 0x00
c:  .word -1
    .text 0x00400000
    la $t0, a
    lb $t1, 0($t0)
    lb $t2, 2($t0)
    add $t3, $t1, $t2
    sb $t3, 3($t0)
    add $t1, $0, $0
    .end
```

Read instruction at address 0x00400000

The fetched instruction is placed in the Instruction Register (IR)

Instruction 2



```
$1  = 0x10000000
$8  = 0x10002000
```

```
PC  = 0x00400004
IR  = 0x34282000
```

OR operation with [$1] and 8192

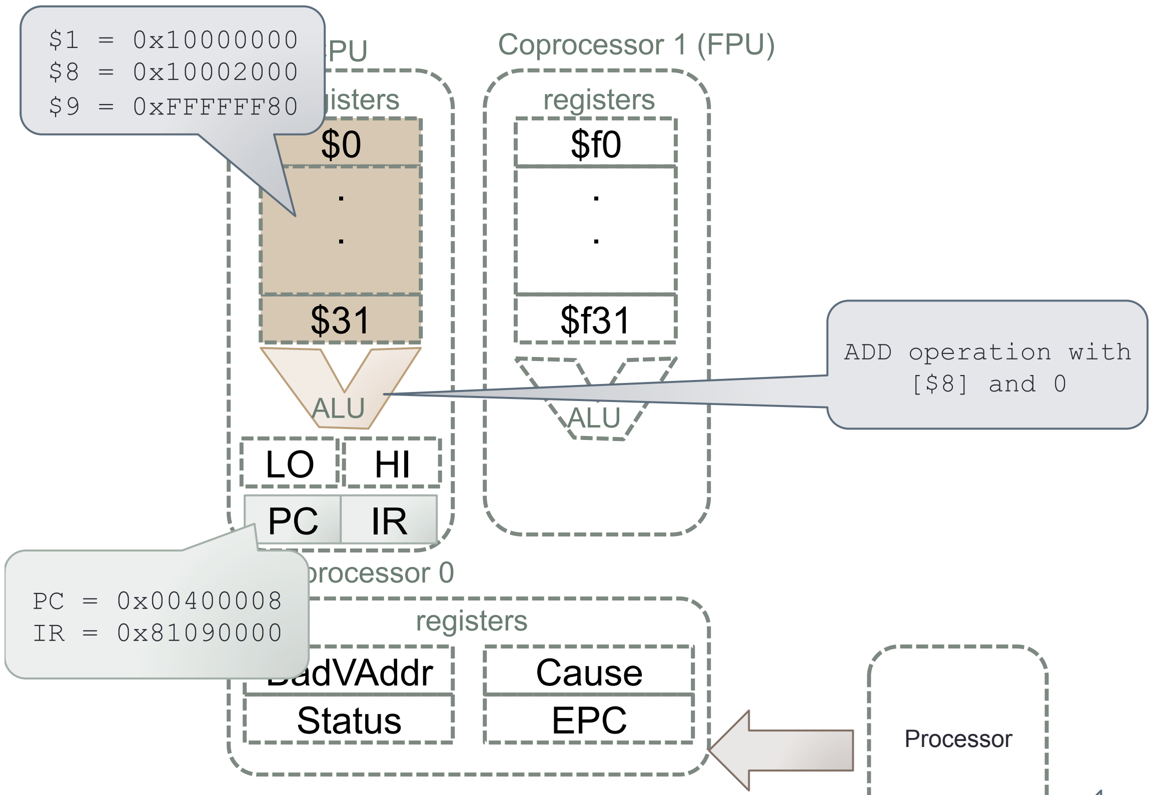
`ori $8, $1, 8192`

Read instruction at address 0x00400004

```
              Contents
          00  0x3c011000
0x00400004    0x34282000
0x00400008    0x81090000
0x0040000C    0x810a0002
0x00400010    0x012a5820
0x00400014    0xa10b0003
0x00400018    0x00004820
0x0040001C    0x3402000a
0x00400020    0x0000000c
...

0x10002000    0x00a0ff80
0x10002004    0xffffffff
...
```

```
    .data 0x10002000
a:  .byte 0x80, 0xFF, 0xA0, 0x00
c:  .word -1
    .text 0x00400000
    la $t0, a
    lb $t1, 0($t0)
    lb $t2, 2($t0)
    add $t3, $t1, $t2
    sb $t3, 3($t0)
    add $t1, $0, $0
    .end
```

Instruction 3



```
$1 = 0x10000000
$8 = 0x10002000
$9 = 0xFFFFFF80
```

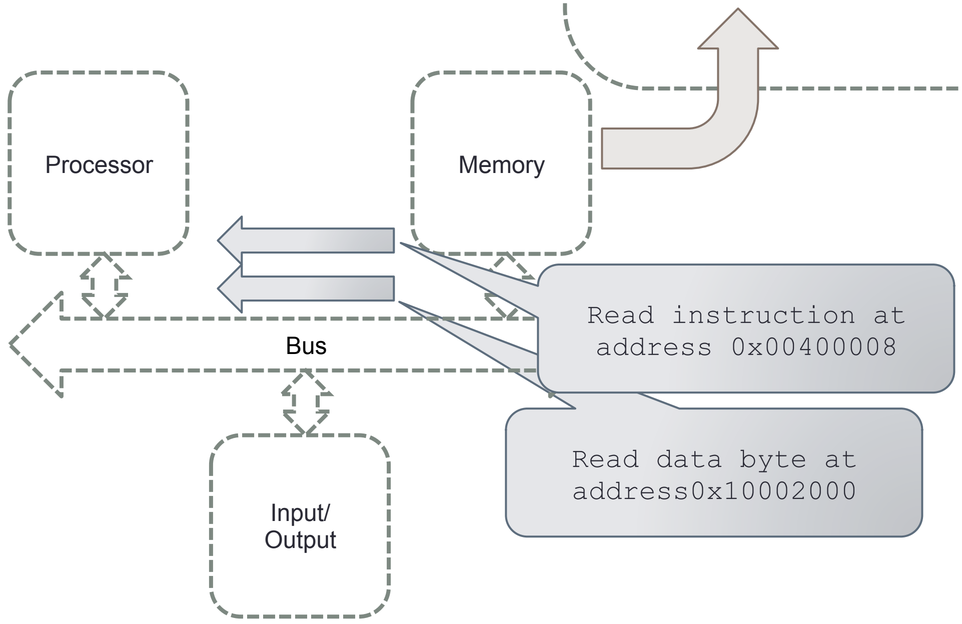
```
PC = 0x00400008
IR = 0x81090000
```

ADD operation with [$8] and 0

Memory contents

`lb $9, 0($8)`

[illegible]	Contents
000	0x3c011000
[illegible]	0x34282000
0x00400008	0x81090000
0x0040000C	0x810a0002
0x00400010	0x012a5820
0x00400014	0xa10b0003
0x00400018	0x00004820
0x0040001C	0x3402000a
0x00400020	0x0000000c
...	
0x10002000	0x00a0ff80
0x10002004	0xffffffff
...	

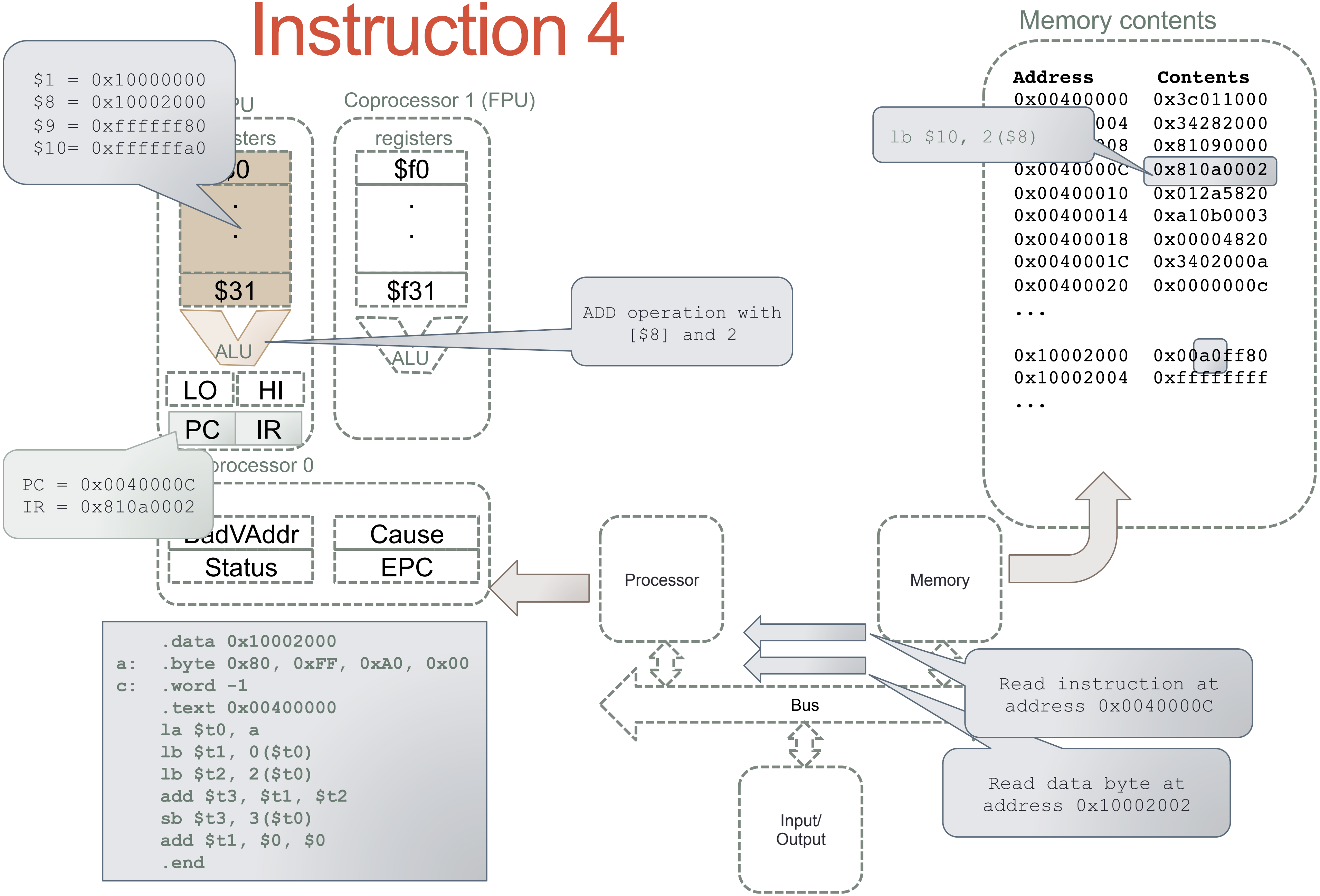


Read instruction at address 0x00400008

Read data byte at address0x10002000

```
    .data 0x10002000
a:  .byte 0x80, 0xFF, 0xA0, 0x00
c:  .word -1
    .text 0x00400000
    la $t0, a
    lb $t1, 0($t0)
    lb $t2, 2($t0)
    add $t3, $t1, $t2
    sb $t3, 3($t0)
    add $t1, $0, $0
    .end
```

Instruction 4



```
$1  = 0x10000000
$8  = 0x10002000
$9  = 0xffffff80
$10= 0xffffffa0
```

```
PC = 0x0040000C
IR = 0x810a0002
```

ADD operation with [$8] and 2

`lb $10, 2($8)`

Memory contents

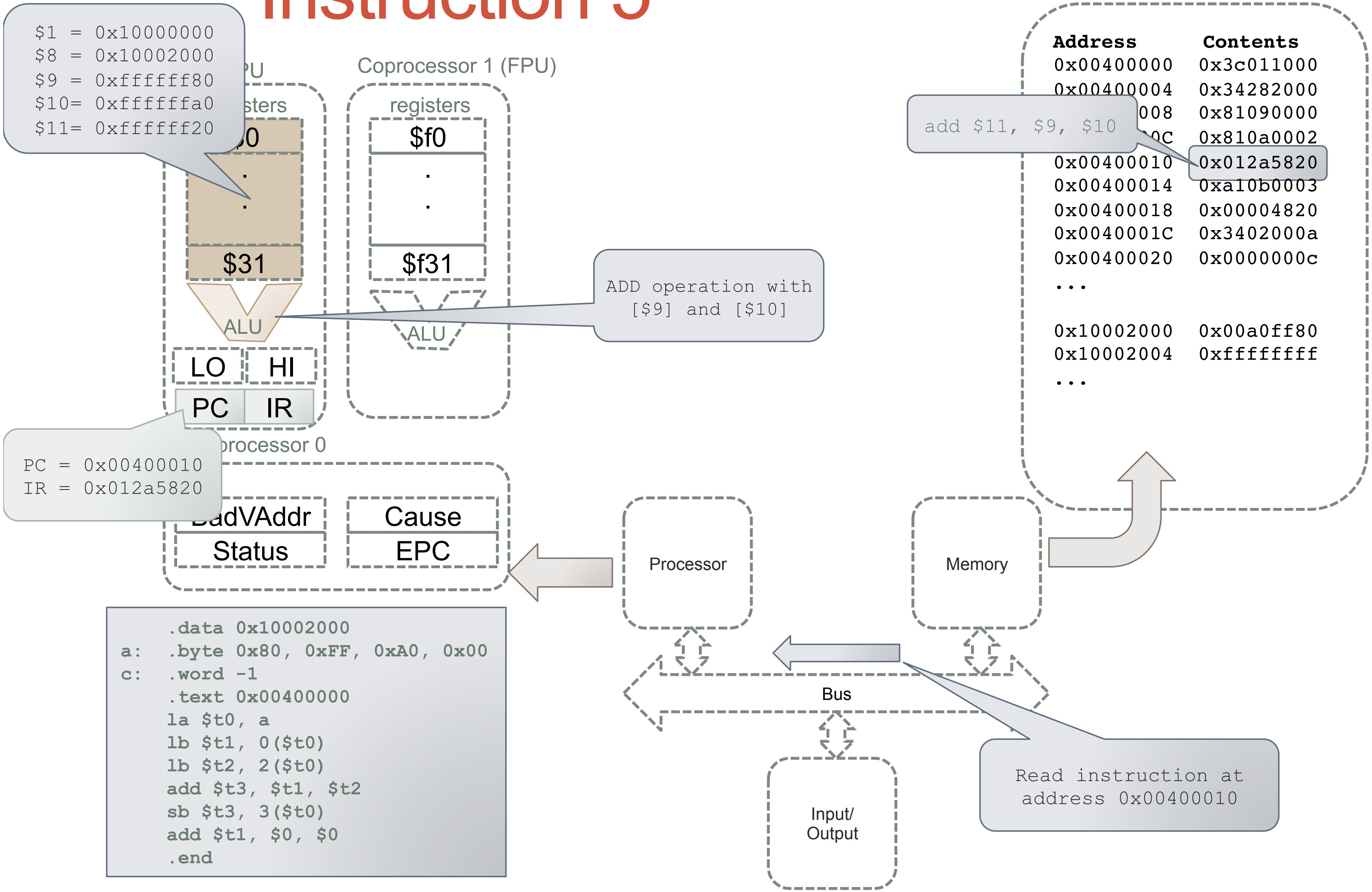
Address	Contents
0x00400000	0x3c011000
004	0x34282000
08	0x81090000
0x0040000C	0x810a0002
0x00400010	0x012a5820
0x00400014	0xa10b0003
0x00400018	0x00004820
0x0040001C	0x3402000a
0x00400020	0x0000000c
...	
0x10002000	0x00a0ff80
0x10002004	0xffffffff
...	

Read instruction at address 0x0040000C

Read data byte at address 0x10002002

```
    .data 0x10002000
a:  .byte 0x80, 0xFF, 0xA0, 0x00
c:  .word -1
    .text 0x00400000
    la $t0, a
    lb $t1, 0($t0)
    lb $t2, 2($t0)
    add $t3, $t1, $t2
    sb $t3, 3($t0)
    add $t1, $0, $0
    .end
```

Instruction 5



```
$1  = 0x10000000
$8  = 0x10002000
$9  = 0xffffff80
$10= 0xffffffa0
$11= 0xffffff20
```

```
PC = 0x00400010
IR = 0x012a5820
```

ADD operation with [$9] and [$10]

add $11, $9, $10

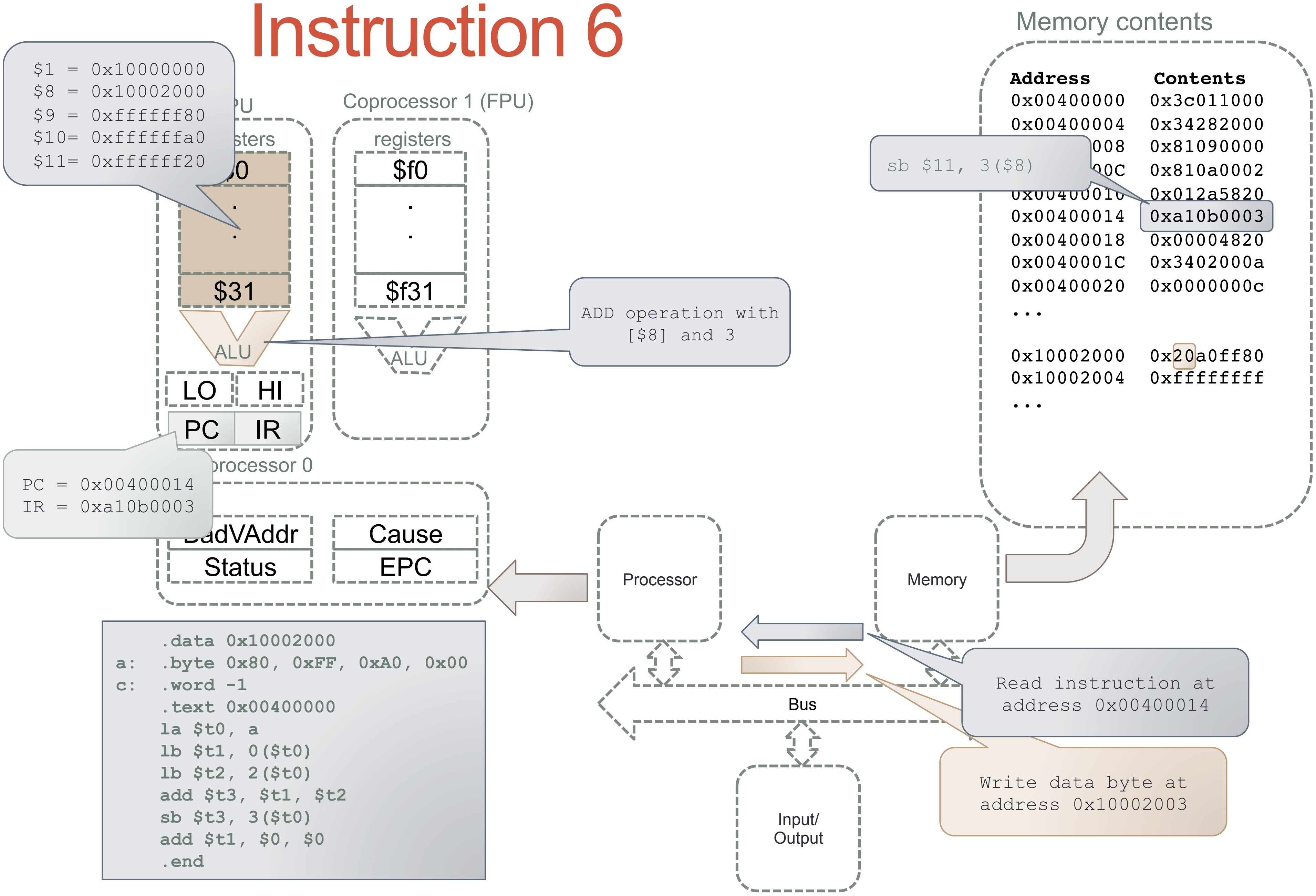
Read instruction at address 0x00400010

Memory contents

Address	Contents
0x00400000	0x3c011000
0x00400004	0x34282000
008	0x81090000
0C	0x810a0002
0x00400010	0x012a5820
0x00400014	0xa10b0003
0x00400018	0x00004820
0x0040001C	0x3402000a
0x00400020	0x0000000c
...	
0x10002000	0x00a0ff80
0x10002004	0xffffffff
...	

```
    .data 0x10002000
a:  .byte 0x80, 0xFF, 0xA0, 0x00
c:  .word -1
    .text 0x00400000
    la $t0, a
    lb $t1, 0($t0)
    lb $t2, 2($t0)
    add $t3, $t1, $t2
    sb $t3, 3($t0)
    add $t1, $0, $0
    .end
```

Instruction 6



```
$1  = 0x10000000
$8  = 0x10002000
$9  = 0xffffff80
$10= 0xffffffa0
$11= 0xffffff20
```

```
PC = 0x00400014
IR = 0xa10b0003
```

sb $11, 3($8)

ADD operation with [$8] and 3

Read instruction at address 0x00400014

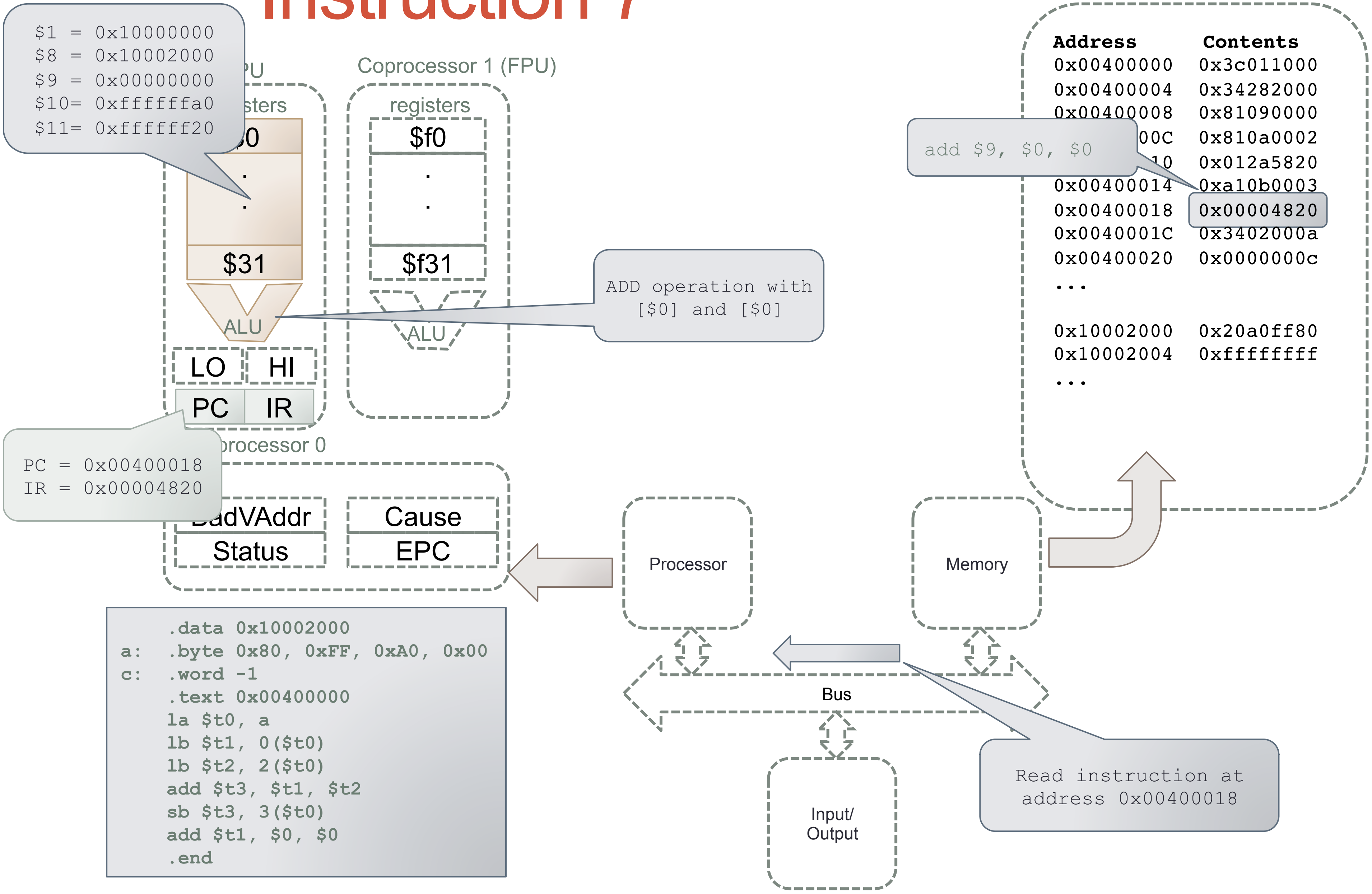
Write data byte at address 0x10002003

Memory contents

Address	Contents
0x00400000	0x3c011000
0x00400004	0x34282000
008	0x81090000
00C	0x810a0002
0x00400010	0x012a5820
0x00400014	0xa10b0003
0x00400018	0x00004820
0x0040001C	0x3402000a
0x00400020	0x0000000c
...	
0x10002000	0x20a0ff80
0x10002004	0xffffffff
...	

```
    .data 0x10002000
a:  .byte 0x80, 0xFF, 0xA0, 0x00
c:  .word -1
    .text 0x00400000
    la $t0, a
    lb $t1, 0($t0)
    lb $t2, 2($t0)
    add $t3, $t1, $t2
    sb $t3, 3($t0)
    add $t1, $0, $0
    .end
```

Instruction 7



```
$1 = 0x10000000
$8 = 0x10002000
$9 = 0x00000000
$10= 0xffffffa0
$11= 0xffffff20
```

```
PC = 0x00400018
IR = 0x00004820
```

ADD operation with [$0] and [$0]

`add $9, $0, $0`

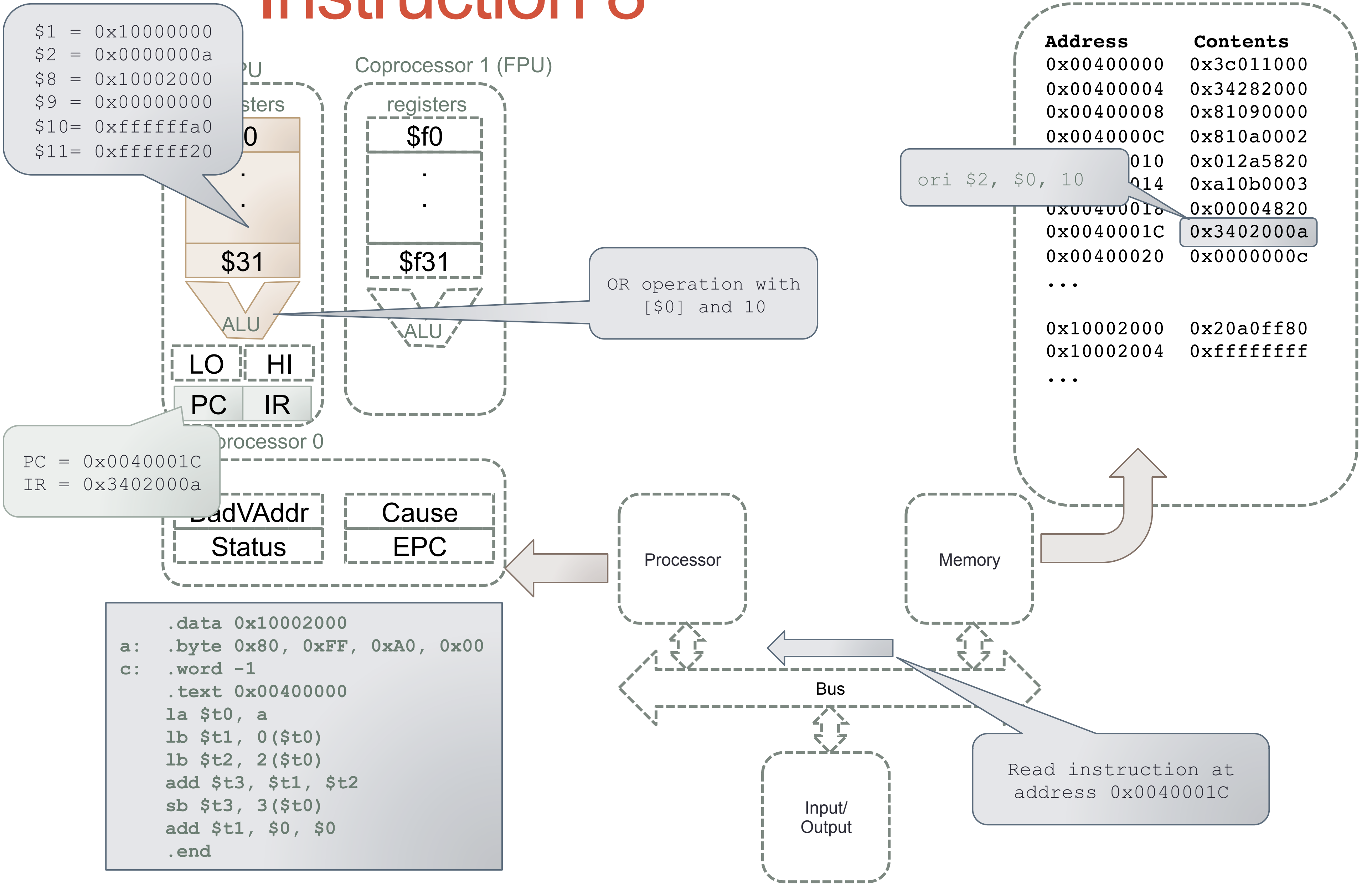
Read instruction at address 0x00400018

Memory contents

Address	Contents
0x00400000	0x3c011000
0x00400004	0x34282000
0x00400008	0x81090000
00C	0x810a0002
10	0x012a5820
0x00400014	0xa10b0003
0x00400018	0x00004820
0x0040001C	0x3402000a
0x00400020	0x0000000c
...	
0x10002000	0x20a0ff80
0x10002004	0xffffffff
...	

```
    .data 0x10002000
a:  .byte 0x80, 0xFF, 0xA0, 0x00
c:  .word -1
    .text 0x00400000
    la $t0, a
    lb $t1, 0($t0)
    lb $t2, 2($t0)
    add $t3, $t1, $t2
    sb $t3, 3($t0)
    add $t1, $0, $0
    .end
```

Instruction 8



```
$1  = 0x10000000
$2  = 0x0000000a
$8  = 0x10002000
$9  = 0x00000000
$10= 0xffffffa0
$11= 0xffffff20
```

```
PC = 0x0040001C
IR = 0x3402000a
```

OR operation with [$0] and 10

Memory contents

```
Address       Contents
0x00400000    0x3c011000
0x00400004    0x34282000
0x00400008    0x81090000
0x0040000C    0x810a0002
          010 0x012a5820
          014 0xa10b0003
0x00400018    0x00004820
0x0040001C    0x3402000a
0x00400020    0x0000000c
...

0x10002000    0x20a0ff80
0x10002004    0xffffffff
...
```

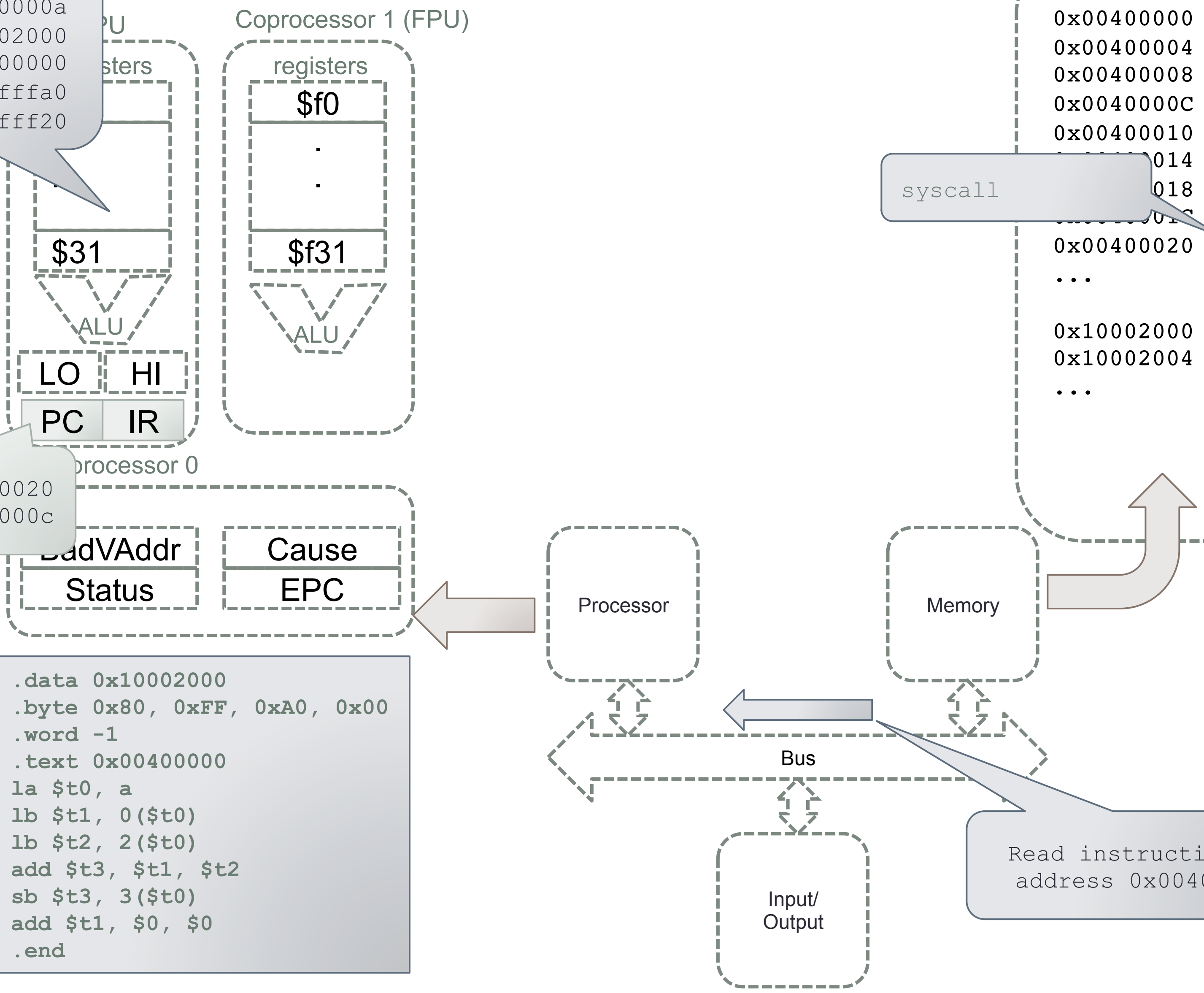
ori $2, $0, 10

```
    .data 0x10002000
a:  .byte 0x80, 0xFF, 0xA0, 0x00
c:  .word -1
    .text 0x00400000
    la $t0, a
    lb $t1, 0($t0)
    lb $t2, 2($t0)
    add $t3, $t1, $t2
    sb $t3, 3($t0)
    add $t1, $0, $0
    .end
```

Read instruction at address 0x0040001C

Instruction 9

```
$1  = 0x10000000
$2  = 0x0000000a
$8  = 0x10002000
$9  = 0x00000000
$10= 0xffffffa0
$11= 0xffffff20
```



```
PC = 0x00400020
IR = 0x0000000c
```

Memory contents

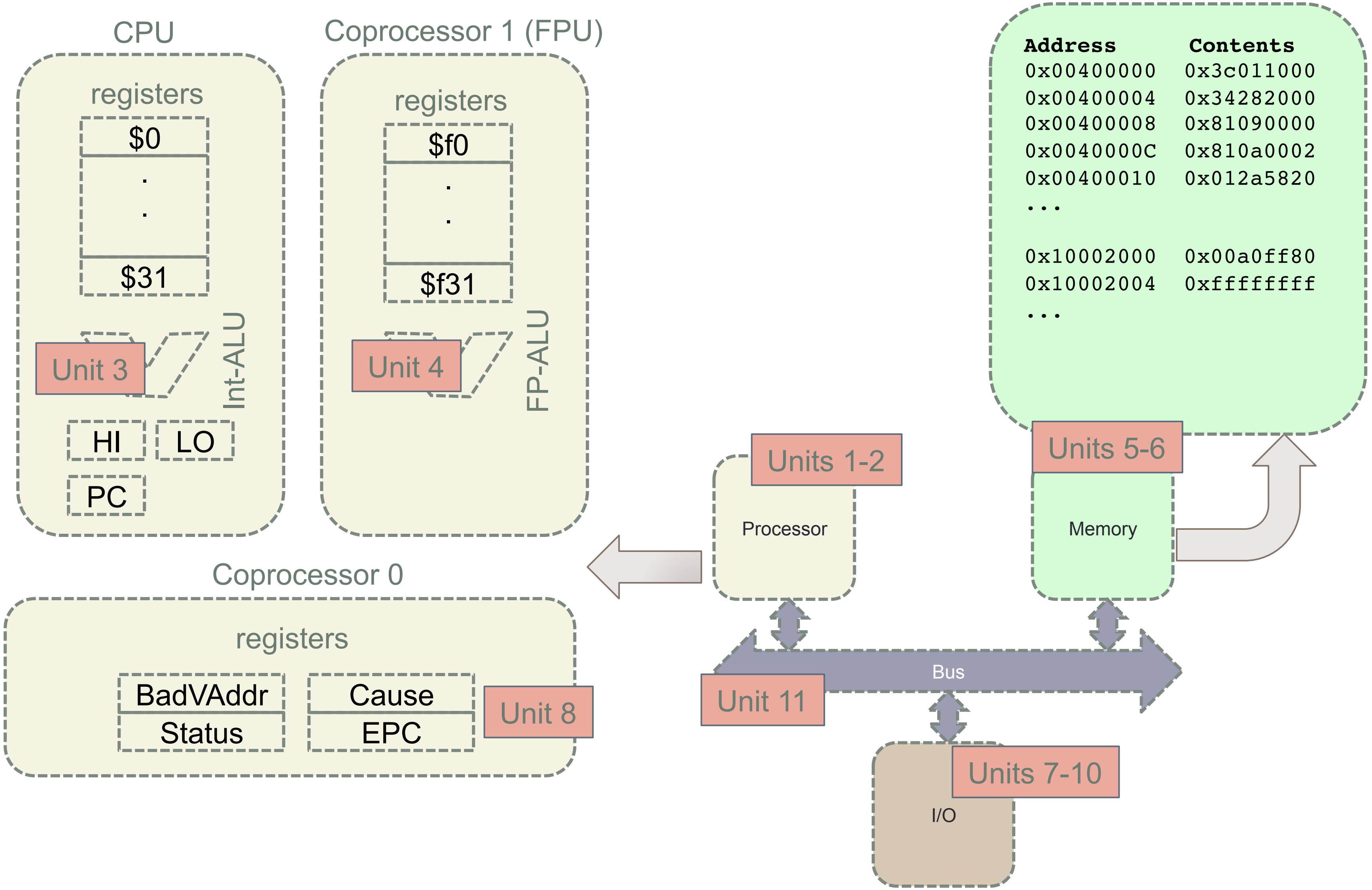
Address	**Contents**
0x00400000	0x3c011000
0x00400004	0x34282000
0x00400008	0x81090000
0x0040000C	0x810a0002
0x00400010	0x012a5820
0x00400014	0xa10b0003
0x00400018	0x00004820
0x0040001C	0x3402000a
0x00400020	0x0000000c
...	
0x10002000	0x20a0ff80
0x10002004	0xffffffff
...	

syscall

```
    .data 0x10002000
a:  .byte 0x80, 0xFF, 0xA0, 0x00
c:  .word -1
    .text 0x00400000
    la $t0, a
    lb $t1, 0($t0)
    lb $t2, 2($t0)
    add $t3, $t1, $t2
    sb $t3, 3($t0)
    add $t1, $0, $0
    .end
```

Read instruction at address 0x00400020

MIPS32 this course



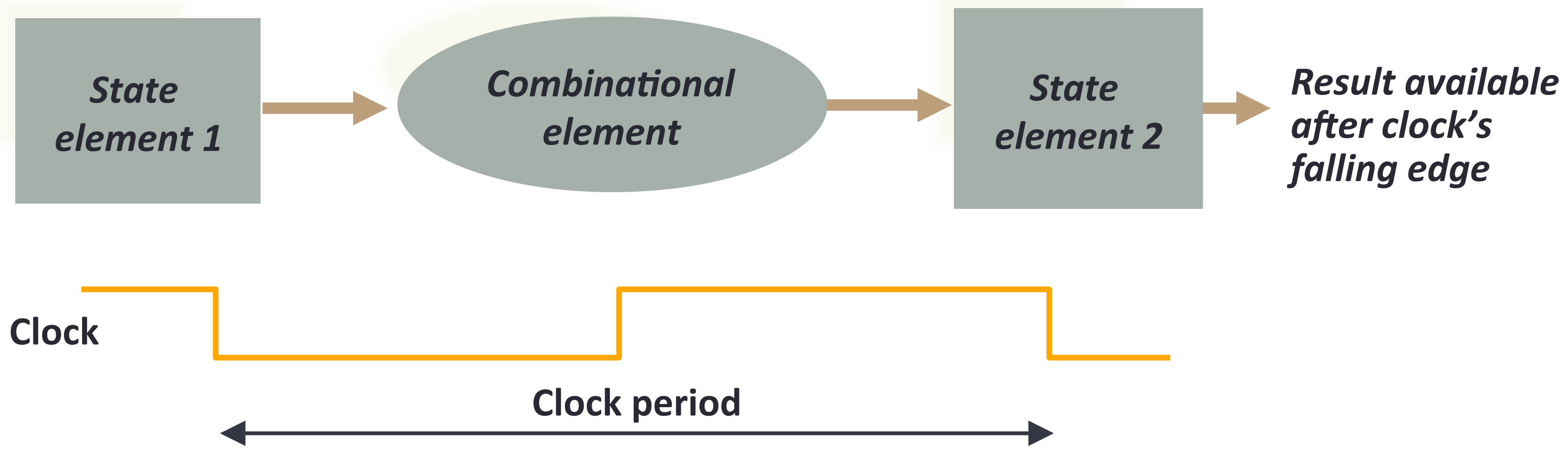
```
Address     Contents
0x00400000  0x3c011000
0x00400004  0x34282000
0x00400008  0x81090000
0x0040000C  0x810a0002
0x00400010  0x012a5820
...

0x10002000  0x00a0ff80
0x10002004  0xffffffff
...
```

2 Datapath design – Introduction

- We will be visiting two processor designs:
  - A monocycle processor (this unit)
    - All instructions execute in a single CPU clock cycle
      - The clock period must accommodate the duration of the slowest instruction
      - The datapath connects functional units point-to-point, it has no *shared* busses
  - A pipelined processor (unit 2)
    - Instructions execute in multiple cycles, but the use of CPU resources is shared among several instructions executing at the same time
      - Hence we can have much shorter clock cycles
- The monocycle design will be the basis for the pipelined version
  - A bus-based datapath is not amenable to *pipelining* (see unit 2)

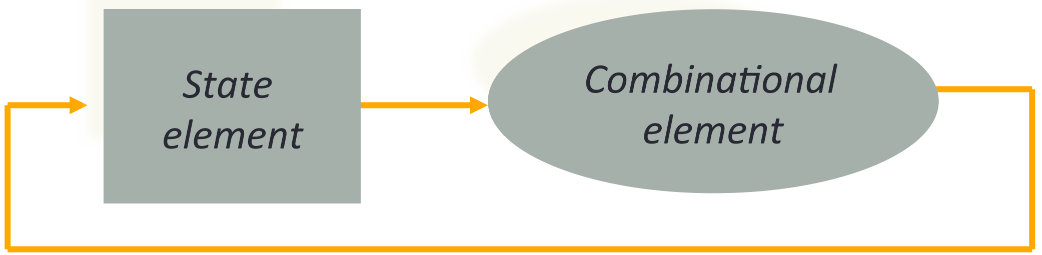
Logic design conventions

- Two kinds of elements
  - Combinational – output depends on current inputs only
    - Eg., the ALU
  - State – output is kept until inputs change AND clock triggers AND write is enabled (if needed)
    - Eg., registers
  - We'll consider edge-triggered clocking

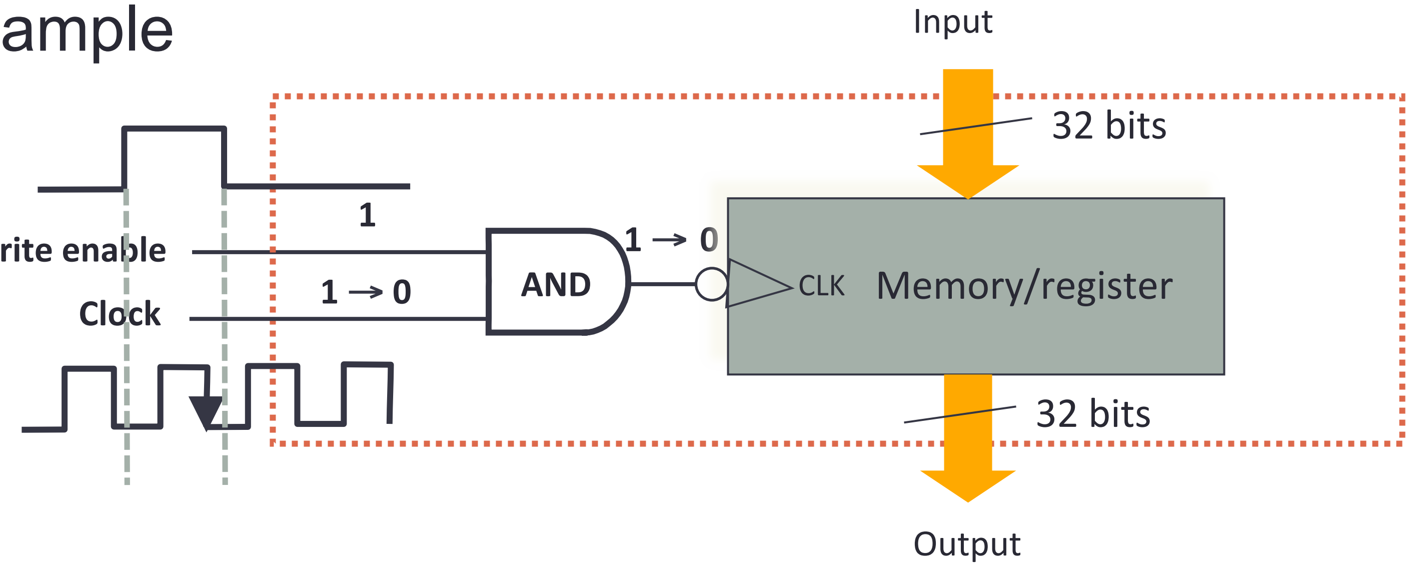


Logic design conventions

- Edge-triggered clocking enables reading and writing a state element in the same clock cycle (eg. a register in the register file)



- Example



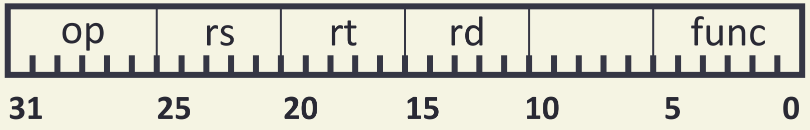
2 Datapath design

- Route to the design
  - Selection of the instruction subset
  - Selection of hardware components
  - Establishing execution phases
  - Complete datapath: putting it all together
    - R-type instructions
    - Memory access instructions
    - Conditional branch (`beq`)

Instruction subset

- We will consider a minimal instruction subset:
  - Arithmetic and logic instructions
    - `add`, `sub`, `and`, `or`, set on less than (`slt`) – all in R and I format
  - Memory instructions
    - Load word and store word (`lw`, `sw`)
  - Conditional branch
    - Branch if equal (`beq`)
  - Unconditional branch
    - Jump (`j`) and register indirect jump (`jr`) – these will be considered in the lab

Type R instructions

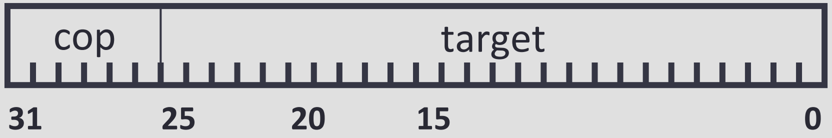


Instruction	Operation
add rd,rs,rt	rd ← rs + rt
sub rd,rs,rt	rd ← rs – rt
and rd,rs,rt	rd ← rs ∧ rt
or rd,rs,rt	rd ← rs ∨ rt

slt rd,rs,rt	**if** rs < rt **then** rd ← 1
	else rd ← 0

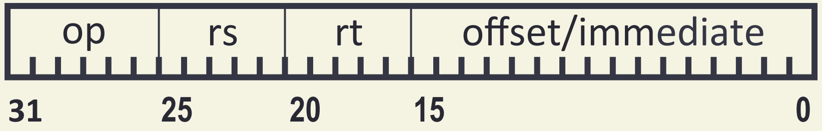
jr rs	CP ← rs

Type J instructions (to be used in a lab session)



Instruction	Operation
j label	CP ← label (CP_{27-0} ← target*4)

Type I instructions



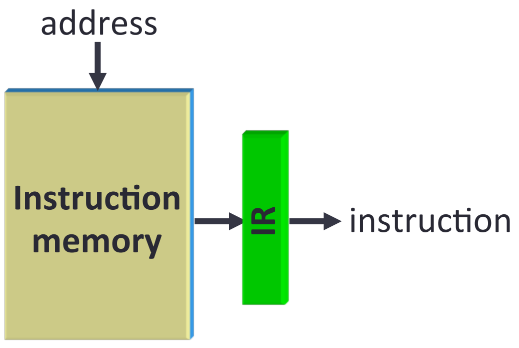
Instruction	Operation
lw rt,offset(rs)	rt ← mem[rs+offset]
sw rt,offset(rs)	mem[rs+offset] ← rt
beq rs,rt,label	**if** rs = rt **then** PC ← label (PC ← PC+ 4 + offset*4)
Instructions with immediate operand	
***<op>* rt,rs,imm**	rt ← rs *op* immediate

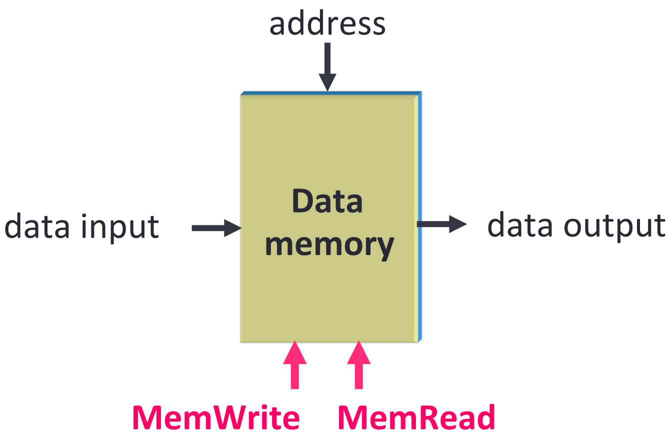
Encoding

Instruction	Format	op	*func* (R-format)
add	R	000000	100000
sub	R	000000	100010
and	R	000000	100100
or	R	000000	100101
slt	R	000000	101010
lw	I	100011	
sw	I	101011	
beq	I	000100	
j	J	000010	
jr	R	000000	001000

Memory

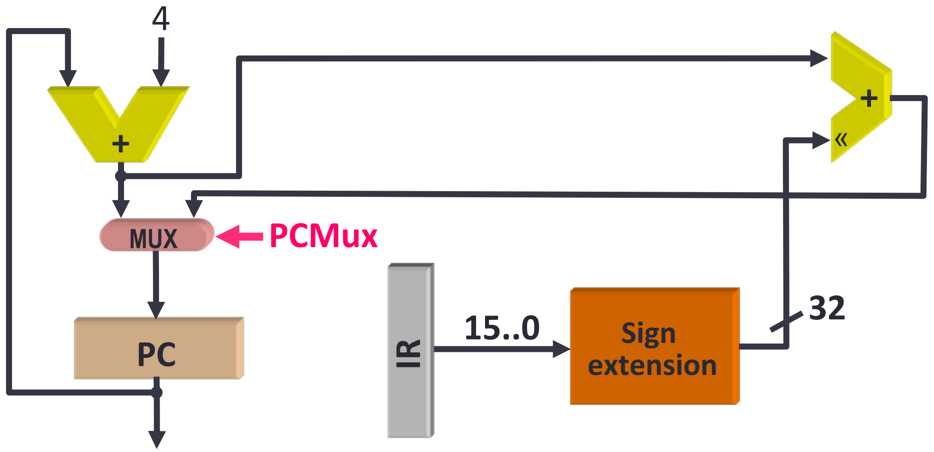
- Single-level hierarchy (ie. no cache), Harvard architecture
  - Instructions (read only)
    - The Instruction Register (IR) latches the current instruction
    - No need for READ signal: I-mem is always read
  - Data (read and write)
- Addressing space: 4 GB
- Bus width: 32 bits





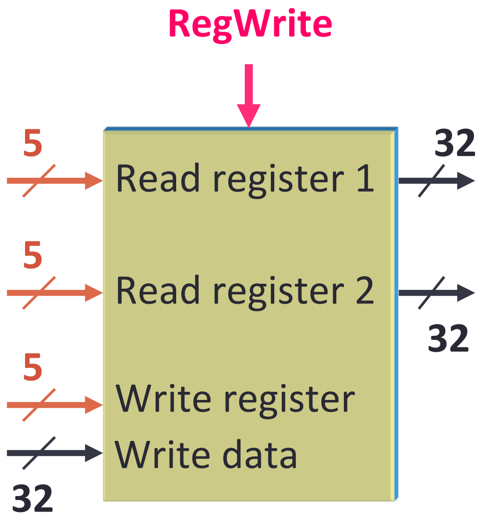
Program counter

- Instructions are fetched from the address pointed to by the *program counter* (PC)
- The PC needs to be incremented by 4 (except for branches)
- For branch instructions, the new PC is PC+4+(offset*4)
  - Offset is encoded in the instruction in 2'sC and must be sign-extended
- A multiplexer enables proper selection of the next PC



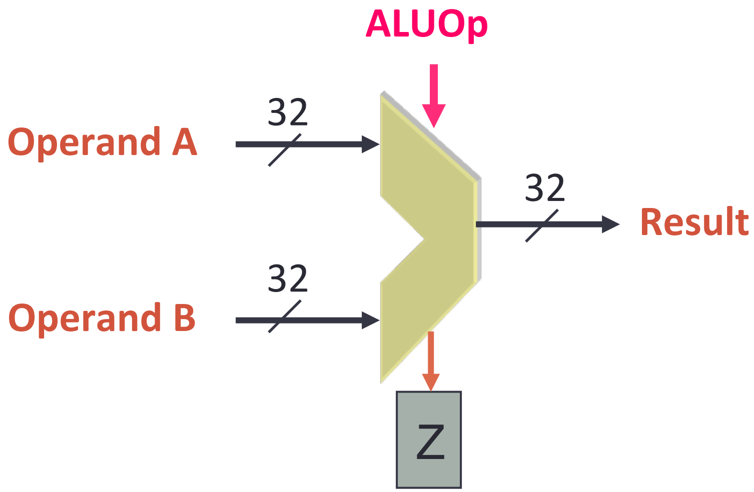
Register file

- Thirty two 32-bit registers ($0 to $31)
- Three ports
  - Two read ports, simultaneous read of two registers
    - Always read two registers – discard if not needed
  - One write port
    - The written register may be one of the read registers



ALU

- The ALU must support the operations of the subset
  - It will calculate addresses for memory instructions as well
- The operation is selected with bits ALUOp
- A zero flag (Z) indicates that the result is zero (for beq)

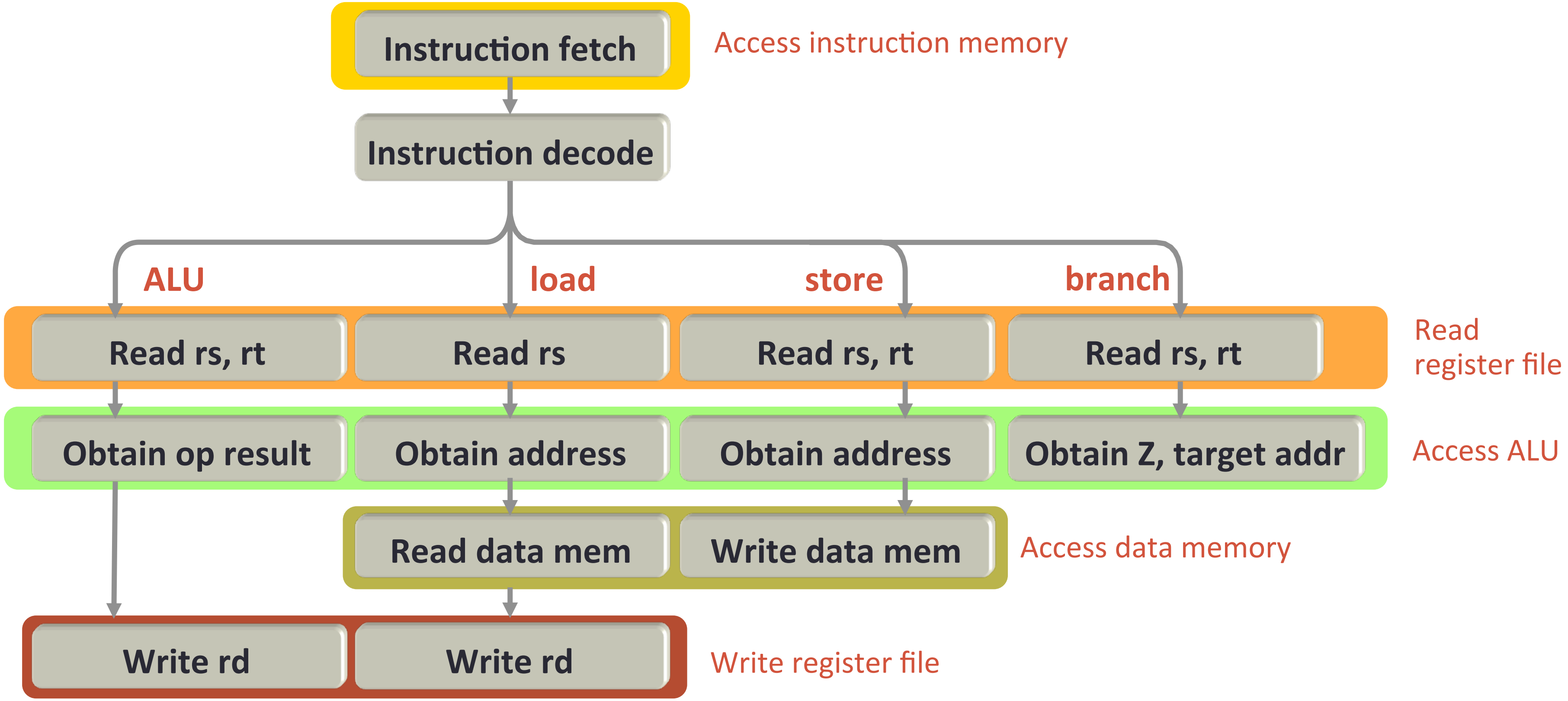


ALUOp	Operation
000	A ∧ B (and)
001	A ∨ B (or)
010	A + B (arithmetic add)
110	A – B (subtraction)
111	A < B (for slt)

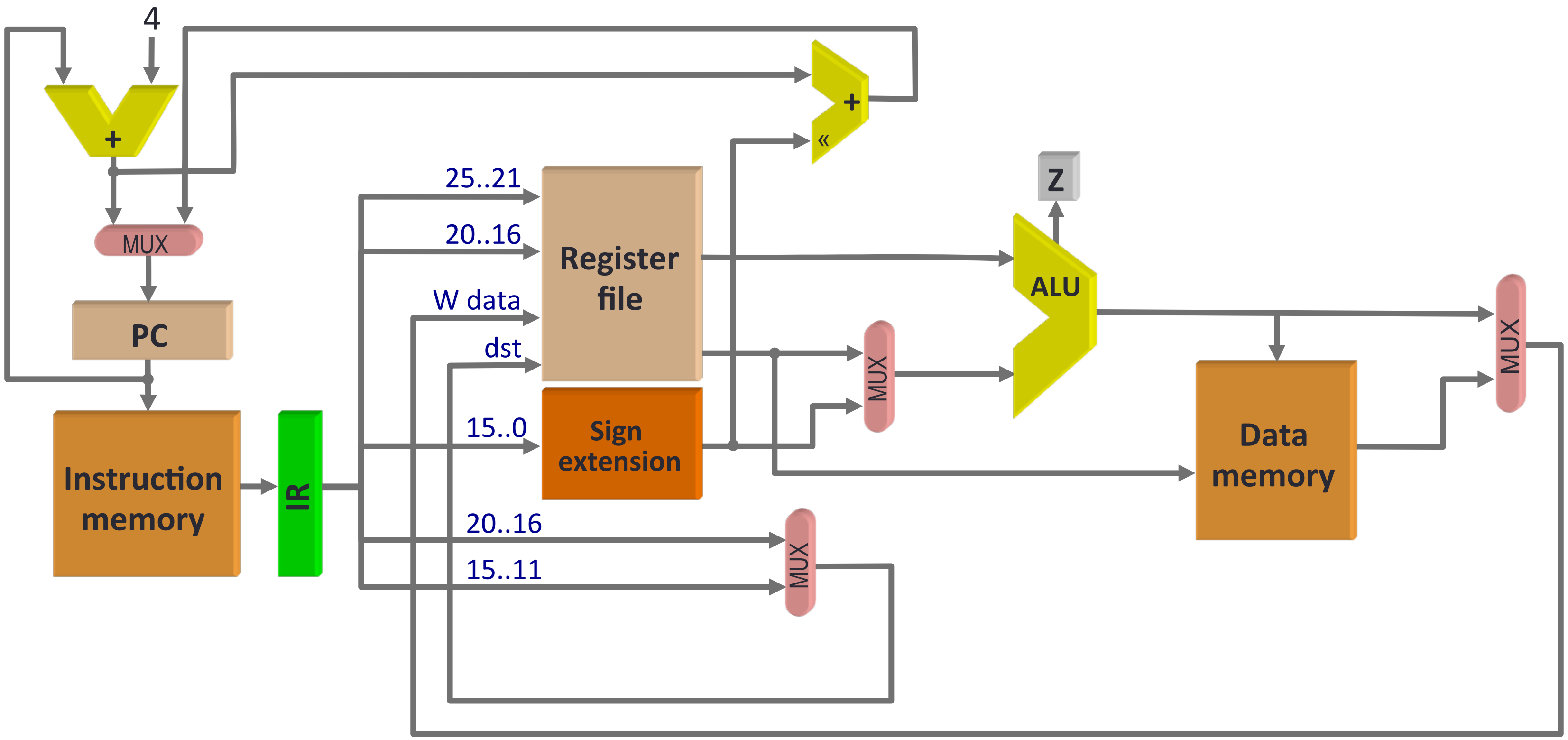
Additional multiplexers

- Three additional multiplexers are needed for selection in the following cases:
  - Operand 2 to ALU – it may be the 2nd source register (R format), or the sign-extended immediate (I format)
  - Destination register number – instruction bits 15..11 (R format) or 20..16 of the instruction (I format)
  - Written data to register file – it may be the ALU output (A/L instructions) or the data memory output (load instruction)

Execution phases



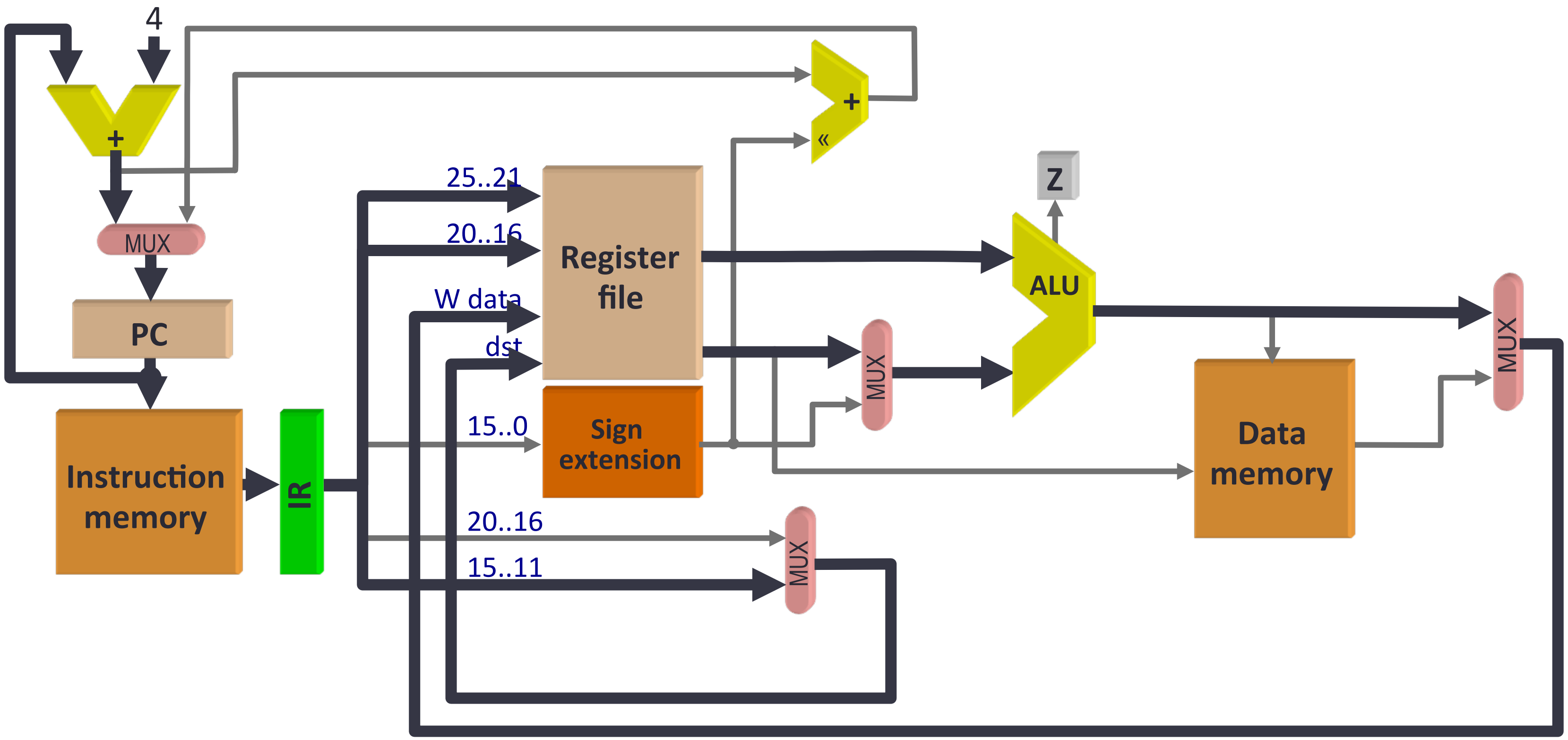
Complete datapath



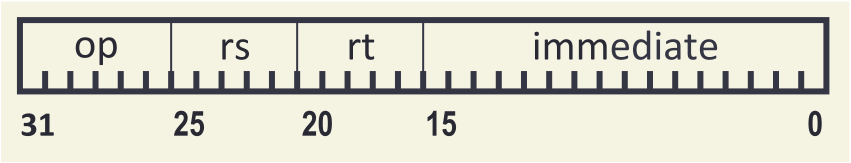
R-type A/L instructions

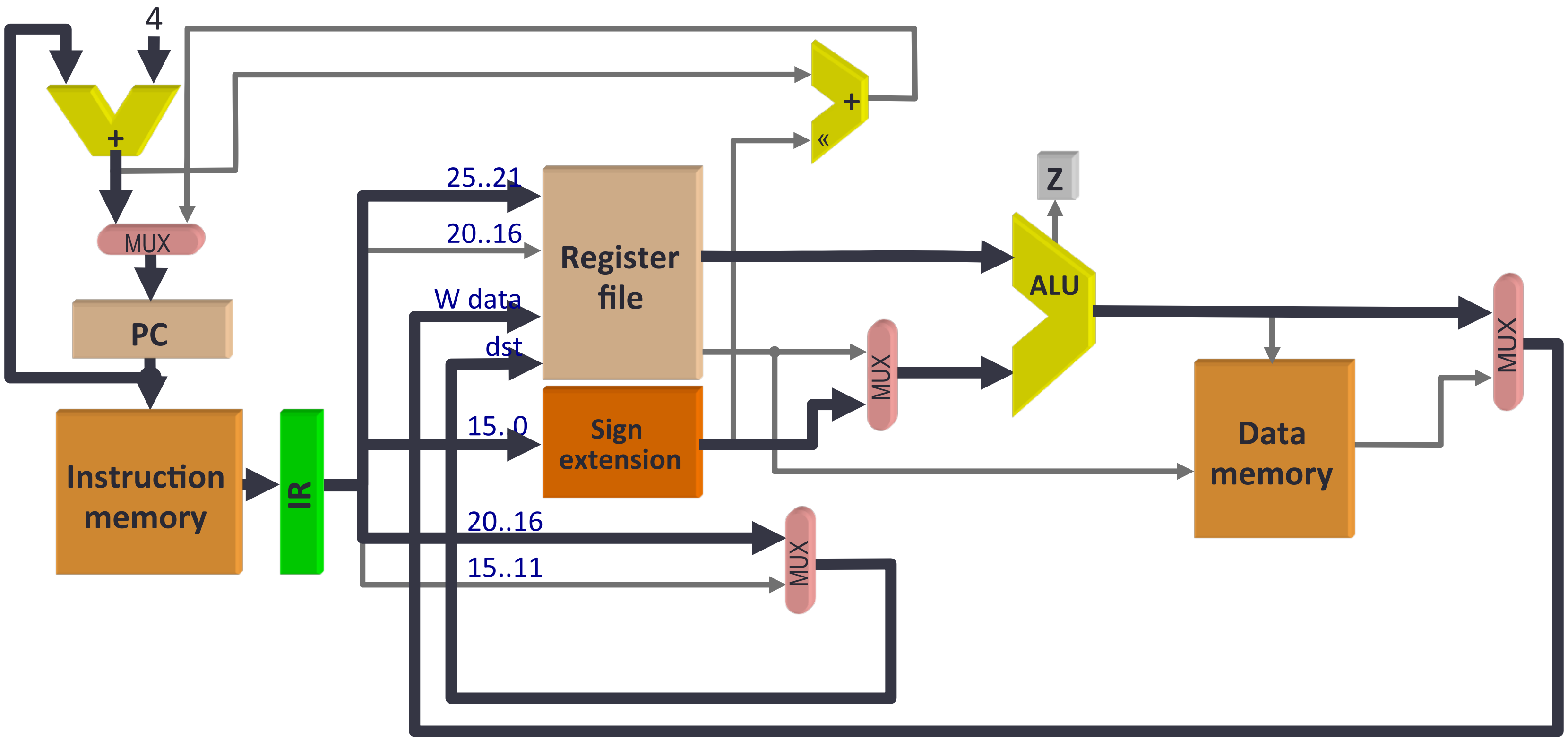
op	rs	rt	rd		func

31 25 20 15 10 5 0

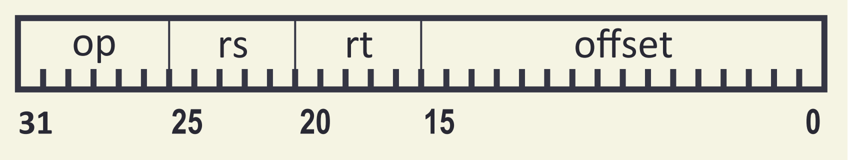


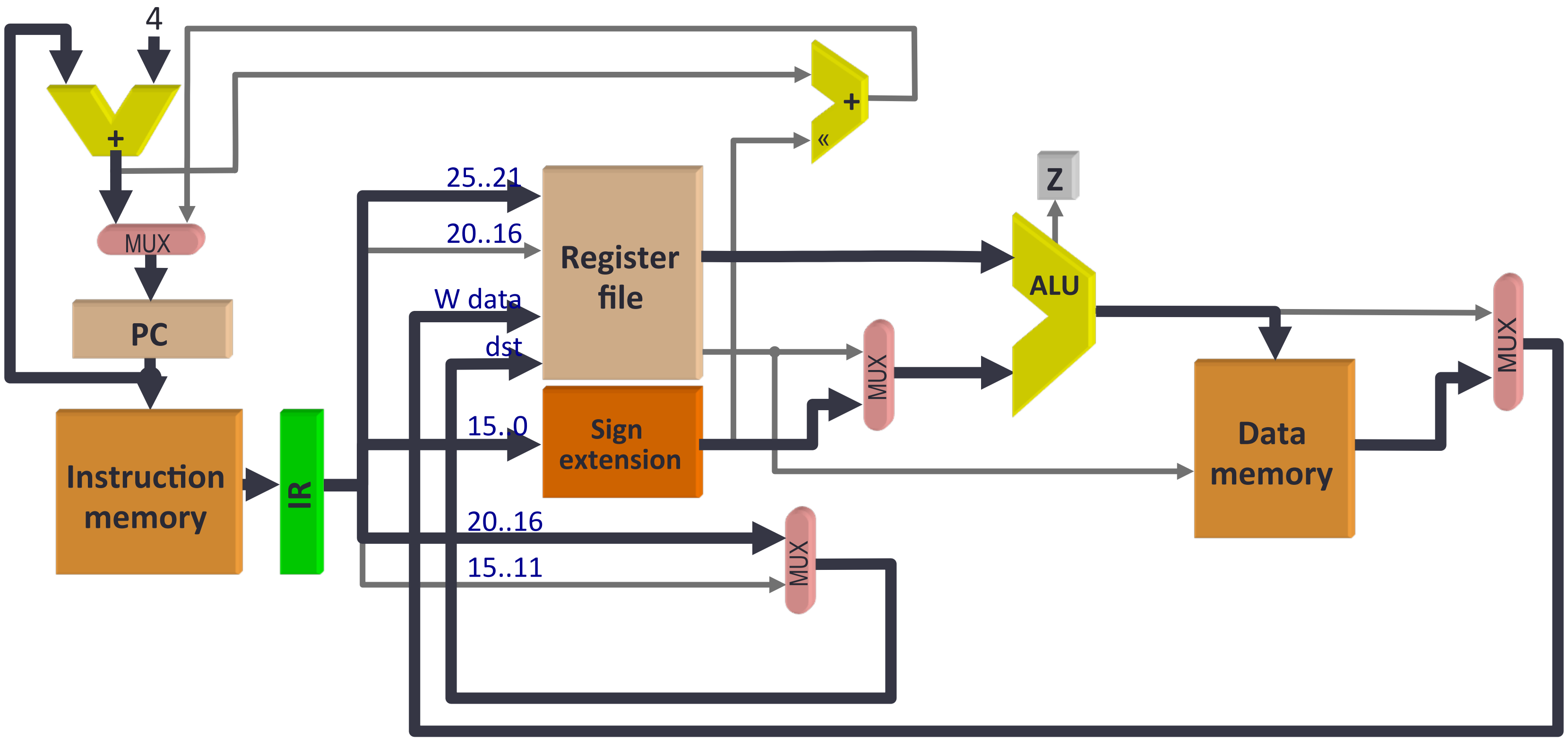
I-type A/L instructions



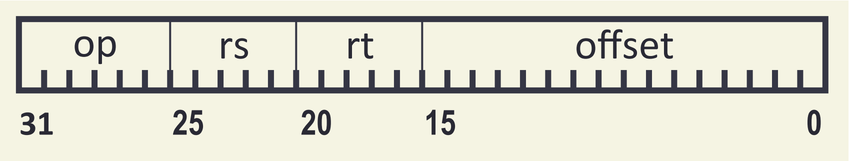


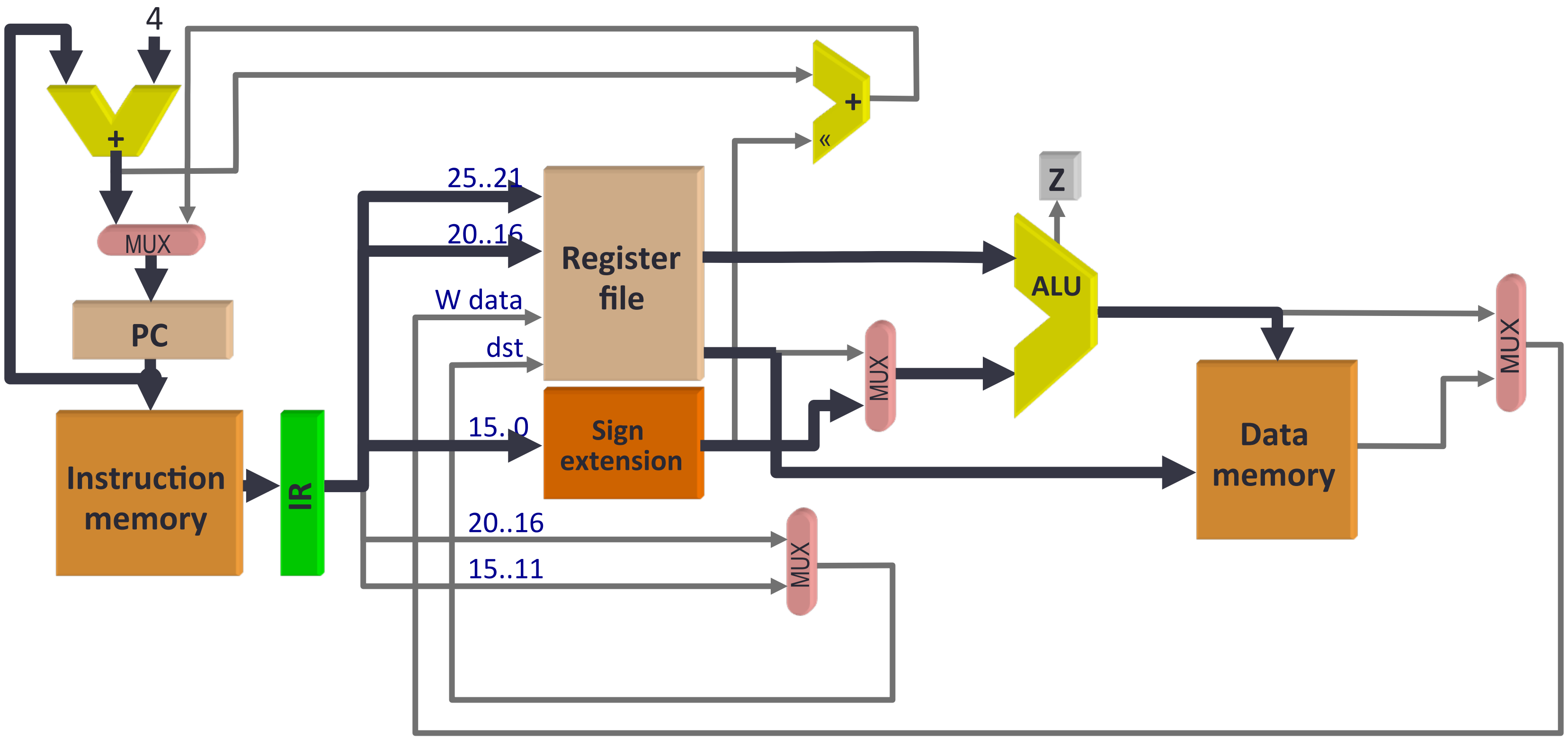
Load word


op
rs
rt
offset
31
25
20
15
0

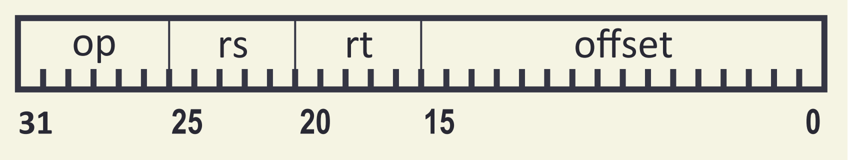

4
+
MUX
PC
Instruction memory
IR
+
«
25..21
20..16
W data
dst
15..0
Register file
Sign extension
Z
ALU
MUX
Data memory
MUX
20..16
15..11
MUX

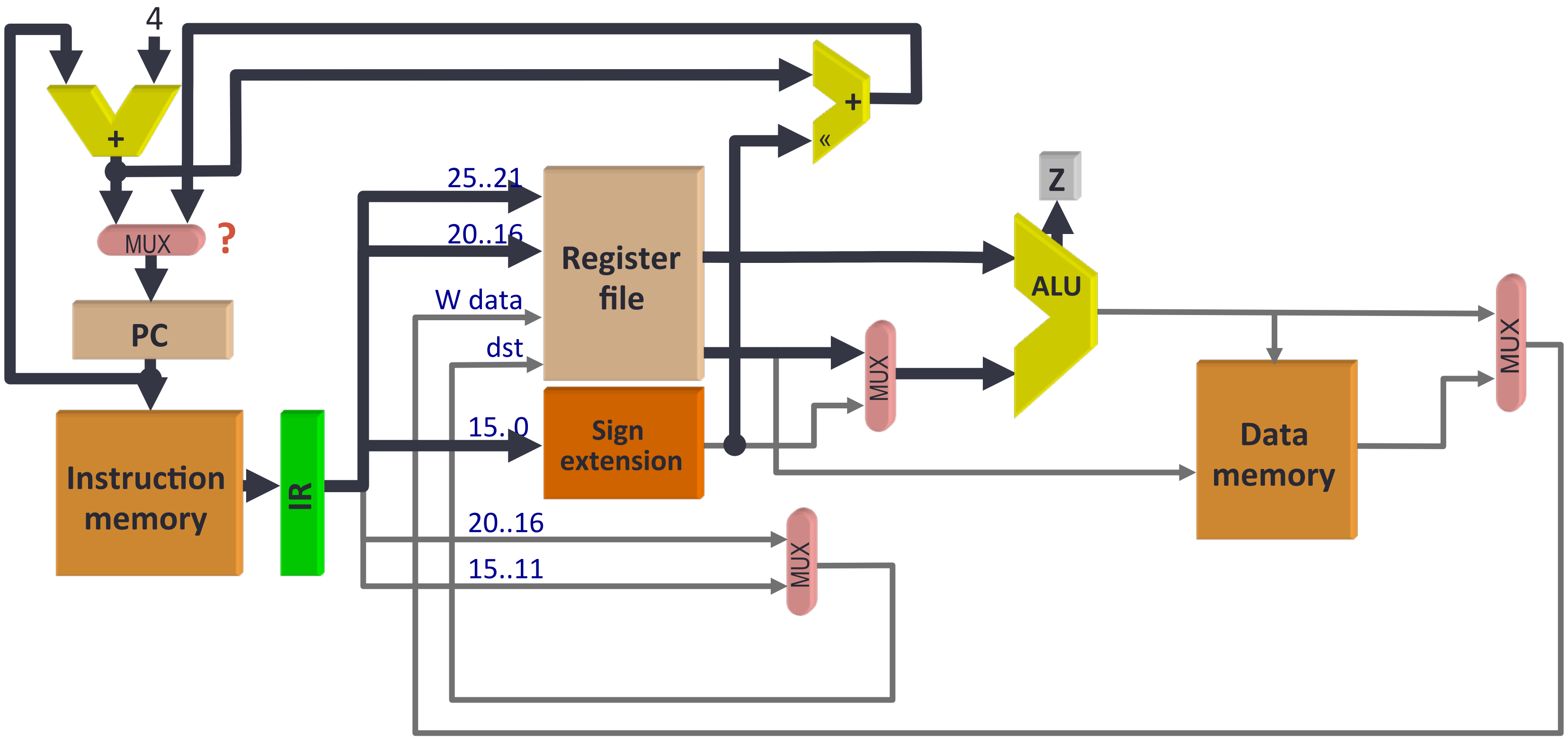
Store word


op
rs
rt
offset
31
25
20
15
0


4
+
MUX
PC
Instruction memory
IR
25..21
20..16
W data
dst
15..0
Register file
Sign extension
20..16
15..11
MUX
+
«
Z
ALU
MUX
Data memory
MUX

Conditional branch (beq)





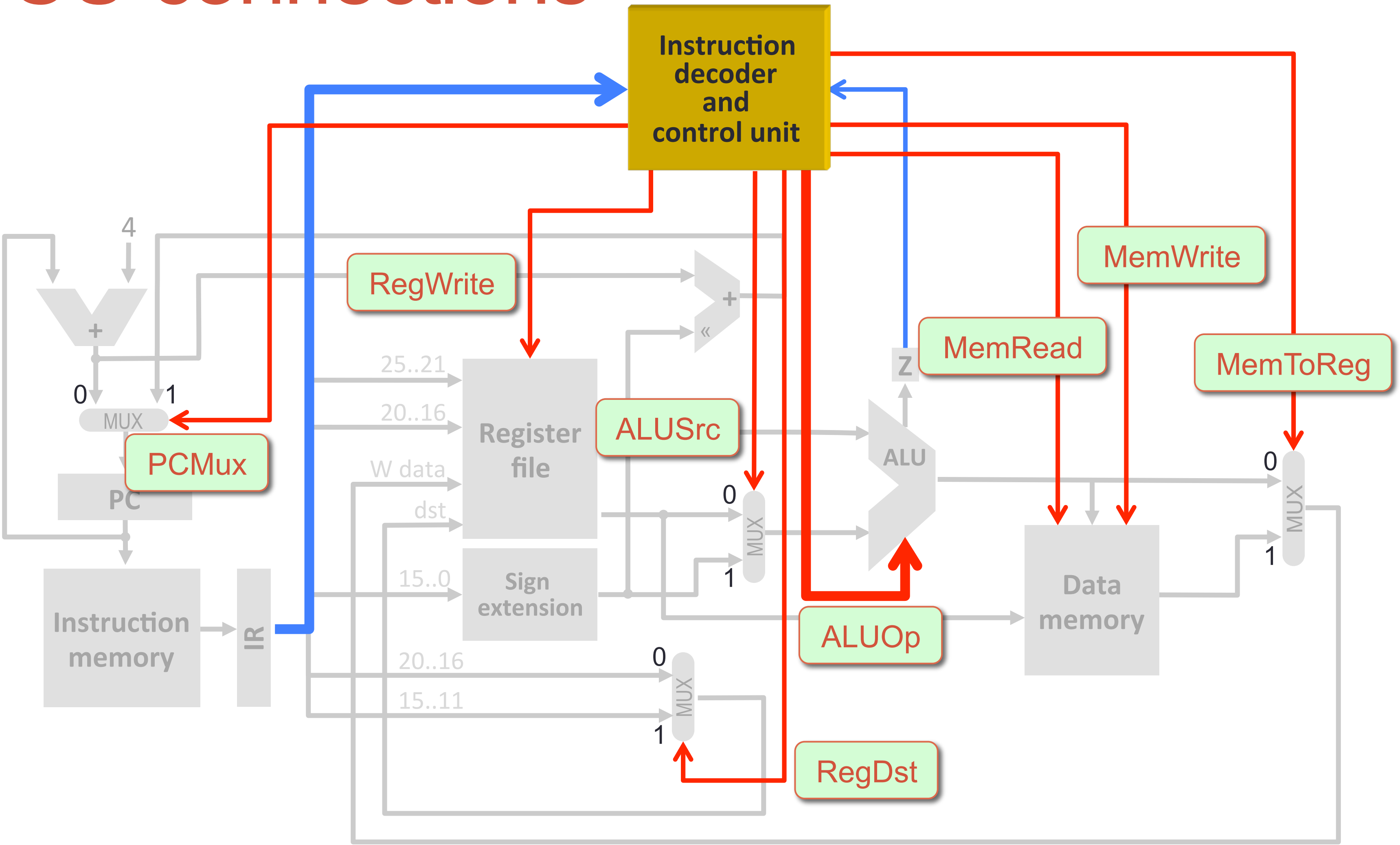
3 Control Unit

- Hardwired CU
- Evaluation
- Other alternatives

Hardwired CU

- The Control Unit (CU) must assert the needed signals in the datapath to enable proper execution of instructions
- The CU needs to decode the current instruction to determine which signals to assert
- The CU can be implemented by a combinational circuit (*hardwired* control unit)
  - Inputs:
    - Bits of IR that enable identification of current instruction (*opcode* / *func*)
    - For beq, the Z flag calculated by the ALU
  - Outputs:
    - All control signals defined so far
      - Mux control (4 signals), register write, memory read & write, ALUOp (3 signals)

CU connections



ALU control signals

ALU specification

ALUOp	Operation
000	A ∧ B (and)
001	A ∨ B (or)
010	A + B (arithmetic add)
110	A – B (subtraction)
111	A < B (for slt)

ALUOp table

		Inputs		Output
Instruction	Format	opcode	func (R)	ALUOp
add	R	000000	100000	0 1 0
sub	R	000000	100010	1 1 0
and	R	000000	100100	0 0 0
or	R	000000	100101	0 0 1
slt	R	000000	101010	1 1 1
lw	I	100011		0 1 0
sw	I	101011		0 1 0
beq	I	000100		1 1 0

Hands on: Complete the ALUOp table by including I-format A/L instructions
(opcodes available in http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html)

All control signals

		Inputs		Outputs							
Instr.	Format	opcode	func (R)	ALUSrc	ALUOp	RegDst	RegWrite	PCMux	MemRead	MemWrite	MemToReg
add	R	000000	100000	0	0 1 0	1	1	0	0	0	0
sub	R	000000	100010	0	1 1 0	1	1	0	0	0	0
and	R	000000	100100	0	0 0 0	1	1	0	0	0	0
or	R	000000	100101	0	0 0 1	1	1	0	0	0	0
slt	R	000000	101010	0	1 1 1	1	1	0	0	0	0
lw	I	100011		1	0 1 0	0	1	0	1	0	1
sw	I	101011		1	0 1 0	X	0	0	0	1	X
beq	I	000100		0	1 1 0	X	0	Z	0	0	X
j	J	000010									
jr	R	000000	001000								

Hands on: Complete this table for j and jr and include also I-format A/L instructions
(opcodes available in http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html)

Evaluation of the single-cycle design

- The datapath+CU is able to execute one instruction per clock cycle. How to determine the clock period?
- We need to know the delays involved in executing every instruction
- Example: assume that…
  - Reading or writing memory costs 2 ns
  - Reading or writing the register file costs 1 ns
  - ALU operates in 2 ns
  - Rest of delays are negligible
- 1. What elements are needed and what’s the overall delay for the execution of each instruction?
- 2. What’s the minimum clock period needed?

Evaluation of the single-cycle design

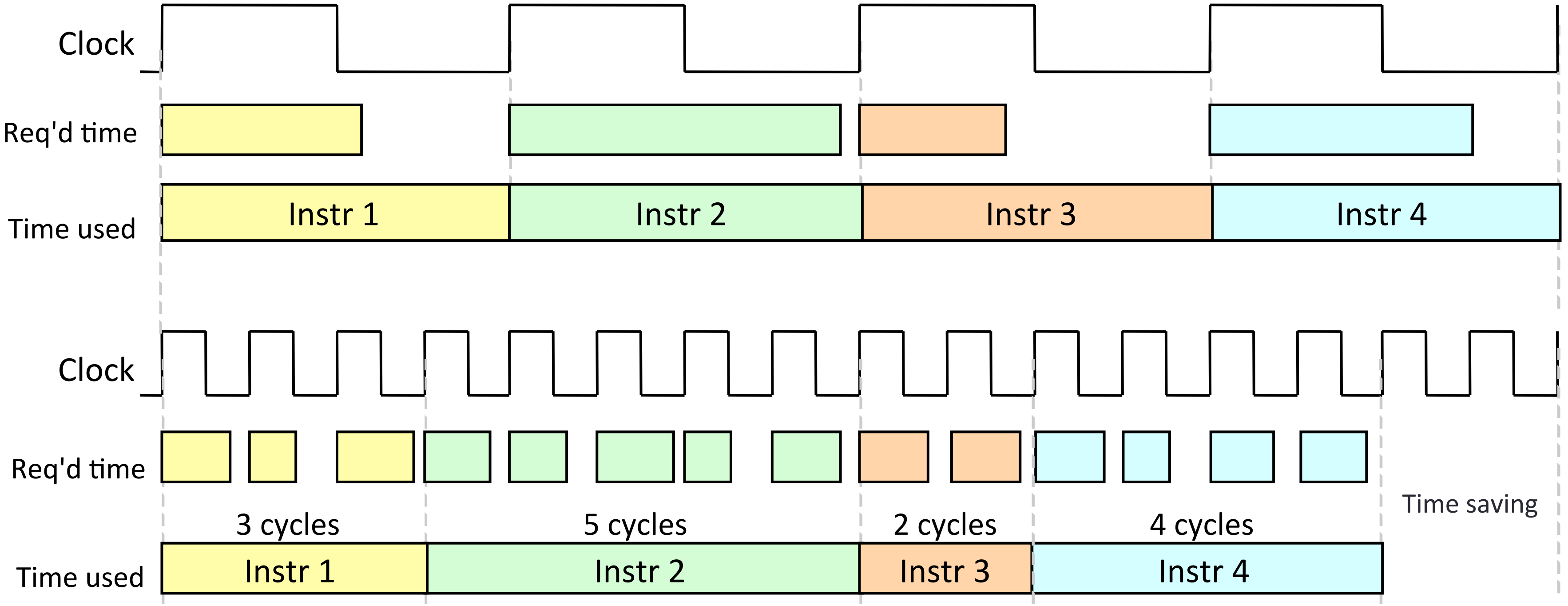
- Answer (1)
  - A/L instructions
    - T = 2 (fetch) + 1 (reg read) + 2 (ALU) + 1 (reg write) = 6 ns
  - Load instruction
    - T = 2 (fetch) + 1 (reg read) + 2 (ALU) + 2 (mem read) + 1 (reg write) = 8 ns
  - Store instruction
    - T = 2 (fetch) + 1 (reg read) + 2 (ALU) + 2 (mem write) = 7 ns
  - Branch
    - T = 2 (fetch) + 1 (reg read) + 2 (ALU) = 5 ns
- Answer (2)
  - We need to select the largest delay as the clock period
    - T_{clk} = 8 ns f_{clk} = 1/8ns = 125 MHz

Evaluation of the single-cycle design

- Although simple and intuitive, this design is inefficient
  - The slowest instruction imposes the clock period
  - No room for improvements in the average case (always assume worst case)
- Single-cycle design has been surpassed by
  - Multi-cycle design (next slide)
  - Pipelining (next unit)

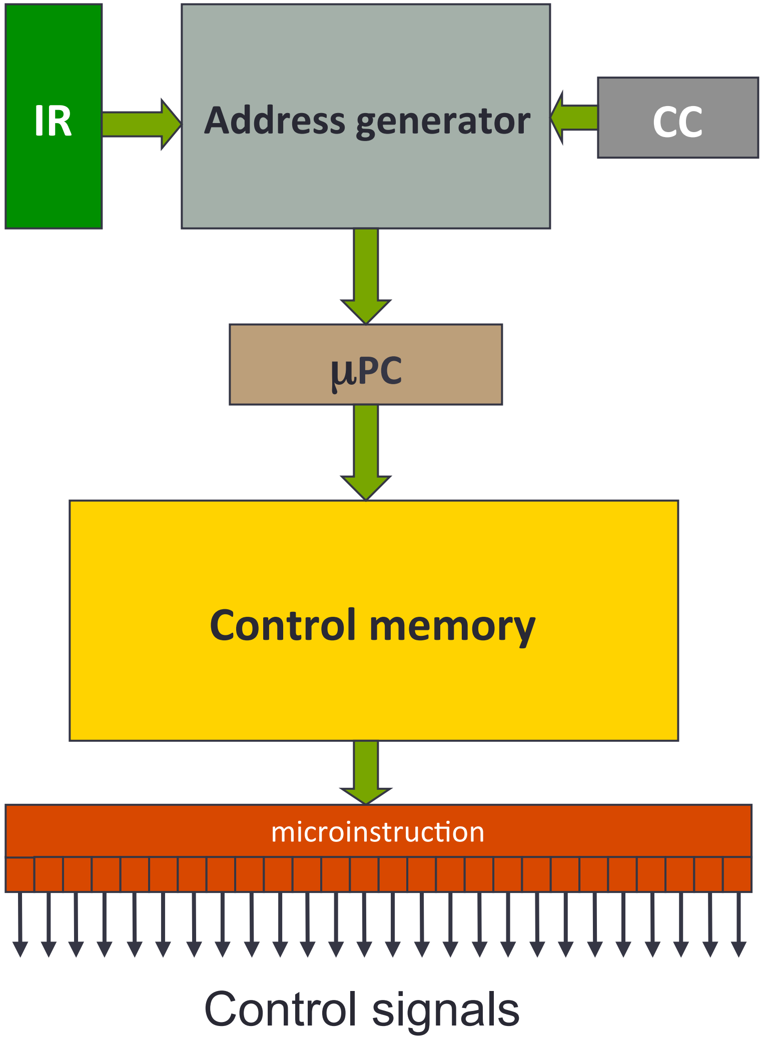
Multi-cycle design

- Selecting the maximum period becomes a worse alternative with more complex instruction sets, exhibiting higher variability in the execution time of instructions
- The multi-cycle design accommodates this variability by giving each instruction the required number of cycles only



Microprogrammed CU

- µP CUs are an alternative when hardwired CUs become too complex, eg. multicycle
- A small, fast control memory contains all control signals for each instruction, for each cycle
  - Words encoding control signals are called microinstructions
- It's not faster than hardwired control, only more flexible
- Often used in the CISC days…



Appendix

- Transfer instructions and pseudoinstructions
  - move, clear, mfhi, mflo, la, li, lui, li.s, li.d, mfc0, mtc0, mfc1, mtc1

```
    .data 0x10000000
a:  .byte 0
    .text 0x00400000

    move $t1, $t0         # $t1 = $t0
    clear $t0             # $t0 = 0

    li $t0, 23            # $t0 = 23
    li.s $f0, 2.45e34     # $f0 = 2.45e34
    li.d $f2, -2.33e11    # $f0|$f1 = -2.33e11

    lui $t0, 0xAABB       # $t0 = 0xAABB0000

    li $t0, 2             # $t0 = 2
    li $t1, 4             # $t1 = 4
    mult $t0, $t1         # hi|lo = 8 (2×4)
    mflo $t0              # $t0 = 8 (low part of result)
    mfhi $t1              # $t1 = 0 (high part of result)
```

```
li $t0, 45            # $t0 = 45
la $t0, a             # $t0 = 0x10000000 (addr of a)

li $t0, 0x33440000    # $t0 = 0x33440000

li.s $f0, 1.0         # $f0 = 1.0 (0x3F800000)
mfc1 $t0, $f0         # $t0 = 0x3F800000 (≠ 1)

li $t0, 1             # $t0 = 1
mtc1 $t0, $f0         # $f0 = 0x00000001 (≠ 1.0)

mfc0 $t0, $12         # $t0 = Status reg of coproc. 0

li $t0, 0             # $t0 = 0
mtc0 $t0, $12         # Status ← 0
```