Lecture Link (Video)
https://web.microsoftstream.com/video/20f4d3a2-05bb-4aea-8c61-afb9ee7091b9

CS222: Computer Architecture

Instructors:
Dr Ahmed Shalaby http://bu.edu.eg/staff/ahmedshalaby14#

الاحترام - الادب - الاخلاق
الطالب - المعيد - الدكتور

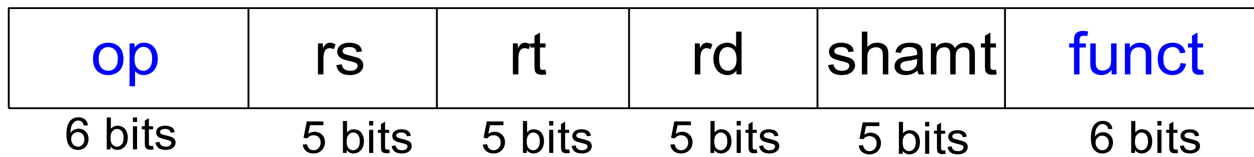
© *Digital Design and Computer Architecture*, 2nd Edition, 2012

ELSEVIER

COMPUTER ARCHITECTURE

Review: Instruction Formats

R-Type



I-Type

J-Type

ELSEVIER

Review: Instruction Formats

R-Type

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Table B.1 Instructions, sorted by opcode

Opcode	Name	Description	Operation
000000 (0)	R-type	all R-type instructions	see Table B.2

Table B.2 R-type instructions, sorted by funct field—Cont'd

Funct	Name	Description	Operation
100000 (32)	add rd, rs, rt	add	[rd] = [rs] + [rt]
100001 (33)	addu rd, rs, rt	add unsigned	[rd] = [rs] + [rt]
100010 (34)	sub rd, rs, rt	subtract	[rd] = [rs] - [rt]
100011 (35)	subu rd, rs, rt	subtract unsigned	[rd] = [rs] - [rt]
100100 (36)	and rd, rs, rt	and	[rd] = [rs] & [rt]
100101 (37)	or rd, rs, rt	or	[rd] = [rs] | [rt]
100110 (38)	xor rd, rs, rt	xor	[rd] = [rs] ^ [rt]
100111 (39)	nor rd, rs, rt	nor	[rd] = ~([rs] | [rt])
101010 (42)	slt rd, rs, rt	set less than	[rs] < [rt] ? [rd] = 1 : [rd] = 0
101011 (43)	sltu rd, rs, rt	set less than unsigned	[rs] < [rt] ? [rd] = 1 : [rd] = 0

COMPUTER ARCHITECTURE

© Digita

EVIER

Review: Instruction Formats

I-Type

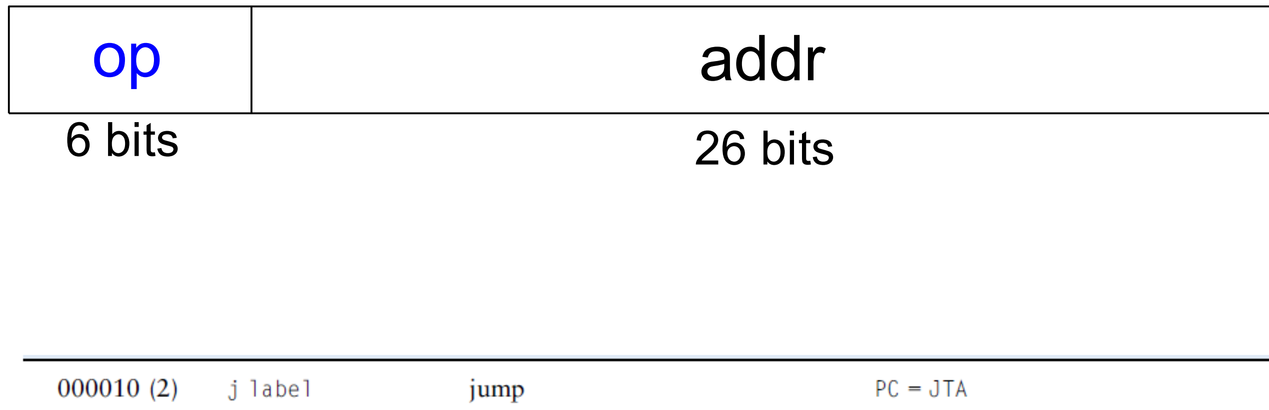
op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

001000 (8)	addi rt, rs, imm	add immediate	[rt] = [rs] + SignImm
001001 (9)	addiu rt, rs, imm	add immediate unsigned	[rt] = [rs] + SignImm
001010 (10)	slti rt, rs, imm	set less than immediate	[rs] < SignImm ? [rt] = 1 : [rt] = 0
001011 (11)	sltiu rt, rs, imm	set less than immediate unsigned	[rs] < SignImm ? [rt] = 1 : [rt] = 0
001100 (12)	andi rt, rs, imm	and immediate	[rt] = [rs] & ZeroImm
001101 (13)	ori rt, rs, imm	or immediate	[rt] = [rs] | ZeroImm
001110 (14)	xori rt, rs, imm	xor immediate	[rt] = [rs] ^ ZeroImm

100000 (32)	lb rt, imm(rs)	load byte	[rt] = SignExt ([Address]$_{7:0}$)
100001 (33)	lh rt, imm(rs)	load halfword	[rt] = SignExt ([Address]$_{15:0}$)
100011 (35)	lw rt, imm(rs)	load word	[rt] = [Address]
100100 (36)	lbu rt, imm(rs)	load byte unsigned	[rt] = ZeroExt ([Address]$_{7:0}$)
100101 (37)	lhu rt, imm(rs)	load halfword unsigned	[rt] = ZeroExt ([Address]$_{15:0}$)

ELSEVIER

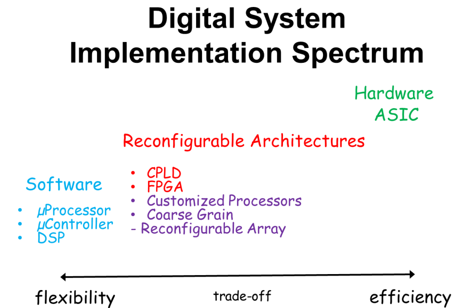
Review: Instruction Formats



ELSEVIER

Power of the Stored Program

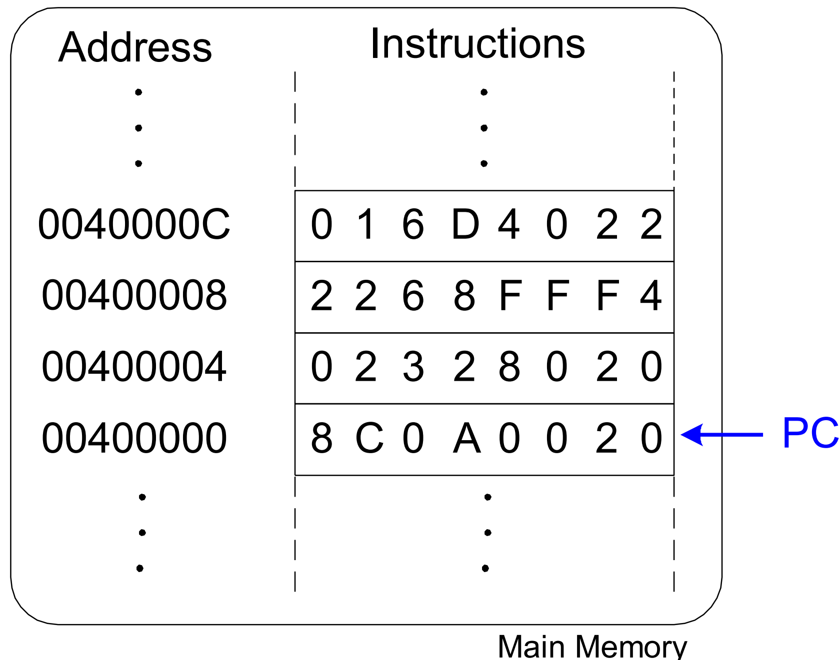
- Sequence of instructions: only difference between two applications
- To run a new program:
  - No rewiring required
  - Simply store new program in memory
- Program Execution:
  - Processor *fetches* (reads) instructions from memory in sequence
  - Processor performs the specified operation
- 32-bit instructions & data stored in memory



ELSEVIER

The Stored Program

Assembly Code	Machine Code
`lw $t2, 32($0)`	`0x8C0A0020`
`add $s0, $s1, $s2`	`0x02328020`
`addi $t0, $s3, -12`	`0x2268FFF4`
`sub $t0, $t3, $t5`	`0x016D4022`



Program Counter (PC): keeps track of current instruction

COMPUTER ARCHITECTURE
ELSEVIER

Interpreting Machine Code

- Start with opcode: tells how to parse rest
- If opcode all 0's
  - R-type instruction
  - Function bits tell operation
- Otherwise
  - opcode tells operation

100000 (32)	add rd, rs, rt	add
100010 (34)	sub rd, rs, rt	subtract

001000 (8)	addi rt, rs, imm	add immediate

$t0-$t7	8-15
$s0-$s7	16-23

Machine Code

	op	rs	rt	imm		
(0x2237FFF1)	001000	10001	10111	1111 1111 1111 0001		
hex	2 2		3 7	F F F 1		
	op	rs	rt	rd	shamt	funct
(0x02F34022)	000000	10111	10011	01000	00000	100010
hex	0 2		F 3	4 0		2 2

Field Values / **Assembly Code**

op	rs	rt	imm			Assembly Code
8	17	23	-15			addi $s7, $s1, -15
op	**rs**	**rt**	**rd**	**shamt**	**funct**	
0	23	19	8	0	34	sub $t0, $s7, $s3

COMPUTER ARCHITECTURE

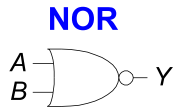
Logical Instructions

- **`and`, `or`, `xor`, `nor`**
  - `and`: useful for **masking** bits
    - Masking all but the least significant byte of a value:

      0xF234012F AND 0x000000FF = 0x0000002F
  - `or`: useful for **combining** bit fields
    - Combine 0xF2340000 with 0x000012BC:

      0xF2340000 OR 0x000012BC = 0xF23412BC
  - `nor`: useful for **inverting** bits:
    - A NOR $0 = NOT A
- **`andi`, `ori`, `xori`**
  - 16-bit immediate is zero-extended (*not* sign-extended)
  - `nori` not needed (`ori` then `NOR$0`)

NOR



$Y = \overline{A + B}$

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

ELSEVIER

Logical Instructions Example 1

Source Registers

$s1	1111	1111	1111	1111	0000	0000	0000	0000
$s2	0100	0110	1010	0001	1111	0000	1011	0111

Assembly Code

```
and $s3, $s1, $s2
or  $s4, $s1, $s2
xor $s5, $s1, $s2
nor $s6, $s1, $s2
```

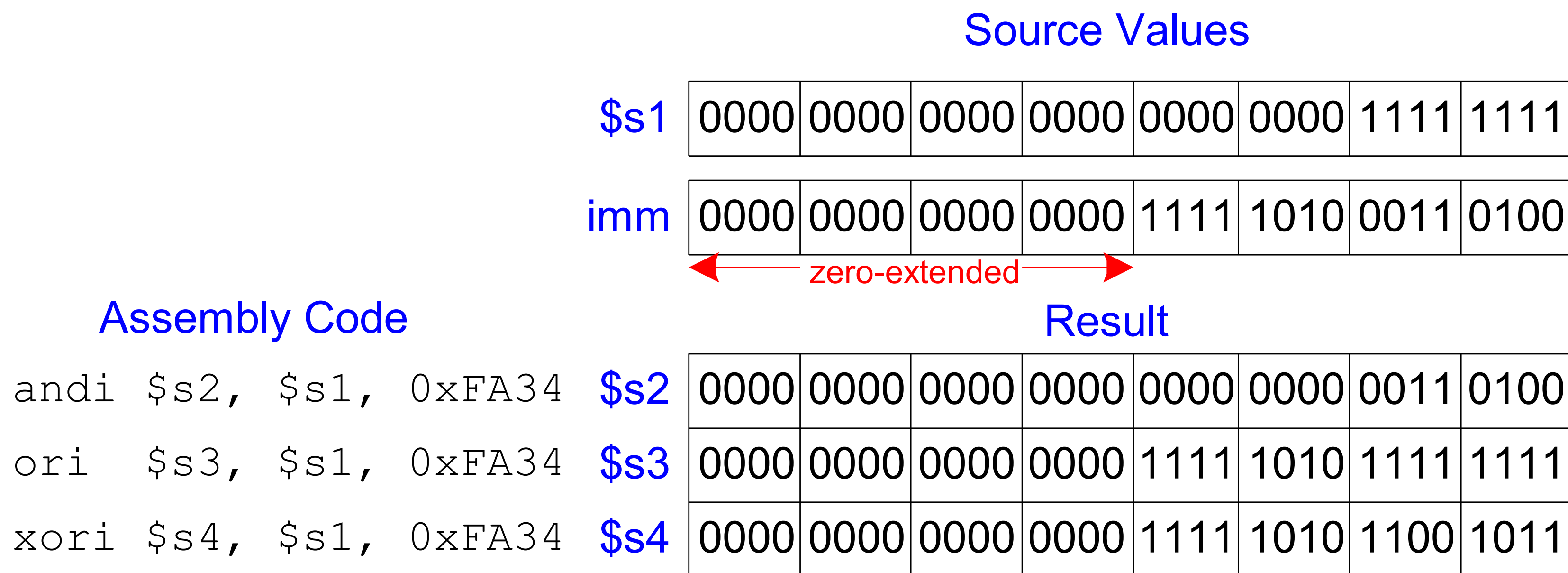
Result

$s3	0100	0110	1010	0001	0000	0000	0000	0000
$s4	1111	1111	1111	1111	1111	0000	1011	0111
$s5	1011	1001	0101	1110	1111	0000	1011	0111
$s6	0000	0000	0000	0000	0000	1111	0100	1000

ELSEVIER

COMPUTER ARCHITECTURE

Logical Instructions Example 2



© *Digital Design and Computer Architecture*, 2nd Edition, 2012

Shift Instructions

- `sll`: shift left logical
  - **Example:** `sll $t0, $t1, 5  # $t0 <= $t1 << 5`
- `srl`: shift right logical
  - **Example:** `srl $t0, $t1, 5  # $t0 <= $t1 >> 5`
- `sra`: shift right arithmetic
  - **Example:** `sra $t0, $t1, 5  # $t0 <= $t1 >>> 5`

© *Digital Design and Computer Architecture*, 2nd Edition, 2012

ELSEVIER

Variable Shift Instructions

- `sllv:` shift left logical variable
  - **Example:** `sllv $t0, $t1, $t2 # $t0 <= $t1 << $t2`
- `srlv:` shift right logical variable
  - **Example:** `srlv $t0, $t1, $t2 # $t0 <= $t1 >> $t2`
- `srav:` shift right arithmetic variable
  - **Example:** `srav $t0, $t1, $t2 # $t0 <= $t1 >>> $t2`

Table B.2 R-type instructions, sorted by funct field

Funct	Name	Description	Operation
000000 (0)	sll rd, rt, shamt	shift left logical	[rd] = [rt] << shamt
000010 (2)	srl rd, rt, shamt	shift right logical	[rd] = [rt] >> shamt
000011 (3)	sra rd, rt, shamt	shift right arithmetic	[rd] = [rt] >>> shamt
000100 (4)	sllv rd, rt, rs	shift left logical variable	[rd] = [rt] << $[rs]_{4:0}$
000110 (6)	srlv rd, rt, rs	shift right logical variable	[rd] = [rt] >> $[rs]_{4:0}$
000111 (7)	srav rd, rt, rs	shift right arithmetic variable	[rd] = [rt] >>> $[rs]_{4:0}$

Shift Instructions

Assembly Code / Field Values

Assembly Code	op	rs	rt	rd	shamt	funct
`sll $t0, $s1, 2`	0	0	17	8	2	0
`srl $s2, $s1, 2`	0	0	17	18	2	2
`sra $s3, $s1, 2`	0	0	17	19	2	3
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Machine Code

op	rs	rt	rd	shamt	funct	
000000	00000	10001	01000	00010	000000	(0x00114080)
000000	00000	10001	10010	00010	000010	(0x00119082)
000000	00000	10001	10011	00010	000011	(0x00119883)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Table

Funct	Name
000000 (0)	sll rd, rt, shamt
000010 (2)	srl rd, rt, shamt
000011 (3)	sra rd, rt, shamt
000100 (4)	sllv rd, rt, rs
000110 (6)	srlv rd, rt, rs
000111 (7)	srav rd, rt, rs

Name	Regist
$0	0
$at	1
$v0-$v1	2-3
$a0-$a3	4-7
$t0-$t7	8-15
$s0-$s7	16-23
$t8-$t9	24-25

© *Digital Design and Computer Architecture*, 2nd Edition, 2012

Multiplication, Division

- Special registers: `lo`, `hi`
- 32 × 32 multiplication, 64 bit result
  - `mult $s0, $s1`
  - Result in {`hi`, `lo`}
- 32-bit division, 32-bit quotient, remainder
  - `div $s0, $s1`
  - Quotient in `lo`
  - Remainder in `hi`

011000 (24)	mult rs, rt	multiply	{[hi], [lo]} = [rs] × [rt]
011001 (25)	multu rs, rt	multiply unsigned	{[hi], [lo]} = [rs] × [rt]
011010 (26)	div rs, rt	divide	[lo] = [rs]/[rt], [hi] = [rs]%[rt]
011011 (27)	divu rs, rt	divide unsigned	[lo] = [rs]/[rt], [hi] = [rs]%[rt]

- Moves from `lo`/`hi` special registers
  - `mflo $s2`
  - `mfhi $s3`

ELSEVIER

Branching

- Execute instructions out of sequence
- Types of branches:
  - Unconditional
    - jump (`j`)
    - jump register (`jr`)
    - jump and link (`jal`)
  - Conditional
    - branch if equal (`beq`)
    - branch if not equal (`bne`)

© *Digital Design and Computer Architecture*, 2nd Edition, 2012

ELSEVIER

Review: The Stored Program

Assembly Code	Machine Code
`lw   $t2, 32($0)`	0x8C0A0020
`add  $s0, $s1, $s2`	0x02328020
`addi $t0, $s3, -12`	0x2268FFF4
`sub  $t0, $t3, $t5`	0x016D4022

Stored Program

Address	Instructions
⋮	⋮
0040000C	0 1 6 D 4 0 2 2
00400008	2 2 6 8 F F F 4
00400004	0 2 3 2 8 0 2 0
00400000	8 C 0 A 0 0 2 0 ← PC
⋮	⋮

Main Memory

ELSEVIER

Unconditional Branching (j)

```
# MIPS assembly
  addi $s0, $0, 4              # $s0 = 4
  addi $s1, $0, 1              # $s1 = 1
  j        target              # jump to target
  sra      $s1, $s1, 2         # not executed
  addi     $s1, $s1, 1         # not executed
  sub      $s1, $s1, $s0       # not executed

  target:
  add      $s1, $s1, $s0       # $s1 = 1 + 4 = 5
```

Labels indicate instruction location. They can't be reserved words and must be followed by colon (:)

COMPUTER ARCHITECTURE

ELSEVIER

Conditional Branching (beq)

```
# MIPS assembly
 addi $s0, $0, 4          # $s0 = 0 + 4 = 4
 addi $s1, $0, 1          # $s1 = 0 + 1 = 1
 sll  $s1, $s1, 2         # $s1 = 1 << 2 = 4
 beq  $s0, $s1, target    # branch is taken
 addi $s1, $s1, 1         # not executed
 sub  $s1, $s1, $s0       # not executed

target:                   # label
 add  $s1, $s1, $s0       # $s1 = 4 + 4 = 8
```

COMPUTER ARCHITECTURE

ELSEVIER

COMPUTER ARCHITECTURE

The Branch Not Taken (bne)

```
# MIPS assembly
   addi    $s0, $0, 4          # $s0 = 0 + 4 = 4
   addi    $s1, $0, 1          # $s1 = 0 + 1 = 1
   sll     $s1, $s1, 2         # $s1 = 1 << 2 = 4
   bne     $s0, $s1, target    # branch not taken
   addi    $s1, $s1, 1         # $s1 = 4 + 1 = 5
   sub     $s1, $s1, $s0       # $s1 = 5 – 4 = 1

target:
   add     $s1, $s1, $s0       # $s1 = 1 + 4 = 5
```


ELSEVIER

Unconditional Branching (jr)

```
# MIPS assembly
                                              # $s0  0x00002010
0x00002000          addi $s0, $0, 0x2010
0x00002004          jr   $s0
0x00002008          addi $s1, $0, 1
0x0000200C          sra  $s1, $s1, 2
0x00002010          lw   $s3, 44($s1)
```

`jr` is an **R-type** instruction.

001000 (8)	jr rs	jump register	PC = [rs]

COMPUTER ARCHITECTURE

ELSEVIER

Programming

- High-level languages:
  - e.g., C, Java, Python
  - Written at higher level of abstraction
- Common high-level software constructs:
  - if/else statements
  - for loops
  - while loops
  - arrays
  - function calls

COMPUTER ARCHITECTURE

ELSEVIER

Generating Constants

- 16-bit constants using `addi`:

C Code

```
// int is a 32-bit signed word
int a = 0x4f3c;
```

MIPS assembly code

```
# $s0 = a
addi $s0, $0, 0x4f3c
```

- 32-bit constants using load upper immediate (`lui`) and `ori`:

C Code

```
int a = 0xFEDC8765;
```

MIPS assembly code

```
# $s0 = a
lui $s0, 0xFEDC
ori $s0, $s0, 0x8765
```

lui	F	E	D	C	0	0	0	0
ori	0	0	0	0	8	7	6	5

← zero-extended →

COMPUTER ARCHITECTURE

If Statement

COMPUTER ARCHITECTURE

C Code

```
if (i == j)
  f = g + h;

f = f - i;
```

MIPS assembly code

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
      bne $s3, $s4, L1
      add $s0, $s1, $s2

L1: sub $s0, $s0, $s3
```

Assembly tests opposite case (`i != j`) of high-level code (`i == j`)

ELSEVIER

If/Else Statement

C Code

```
if (i == j)
  f = g + h;
else
  f = f - i;
```

MIPS assembly code

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
      bne $s3, $s4, L1
      add $s0, $s1, $s2
      j   done
L1:   sub $s0, $s0, $s3
done:
```


ELSEVIER

COMPUTER ARCHITECTURE

While Loops

C Code

```
// determines the power
// of x such that 2^x = 128
int pow = 1;
int x   = 0;

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

MIPS assembly code

```
# $s0 = pow, $s1 = x

       addi $s0, $0, 1
       add  $s1, $0, $0
       addi $t0, $0, 128
while: beq  $s0, $t0, done
       sll  $s0, $s0, 1
       addi $s1, $s1, 1
       j    while
done:
```

Assembly tests for the opposite case (`pow == 128`) of the C code (`pow != 128`).

ELSEVIER

COMPUTER ARCHITECTURE

For Loops

```
for (initialization; condition; loop operation)
  statement
```

- `initialization:` executes before the loop begins
- `condition:` is tested at the beginning of each iteration
- `loop operation:` executes at the end of each iteration
- `statement:` executes each time the condition is met

ELSEVIER

For Loops

C Code

```
// add the numbers from 0 to 9
int sum = 0;
int i;

for (i=0; i!=10; i = i+1) {
  sum = sum + i;
}
```

MIPS assembly code

```
# $s0 = i, $s1 = sum
      addi $s1, $0, 0
      add  $s0, $0, $0
      addi $t0, $0, 10
for:  beq  $s0, $t0, done
      add  $s1, $s1, $s0
      addi $s0, $s0, 1
      j    for
done:
```

COMPUTER ARCHITECTURE

ELSEVIER

Less Than Comparison

C Code

```
// add the powers of 2 from 1
// to 100
int sum = 0;
int i;

for (i=1; i < 101; i = i*2) {
  sum = sum + i;
}
```

2^0	1
2^1	2
2^2	4
2^3	8
2^4	16
2^5	32
2^6	64
2^7	128

MIPS assembly code

```
# $s0 = i, $s1 = sum
        addi $s1, $0, 0
        addi $s0, $0, 1
        addi $t0, $0, 101
loop:   slt  $t1, $s0, $t0
        beq  $t1, $0, done
        add  $s1, $s1, $s0
        sll  $s0, $s0, 1
        j    loop
done:
```

$t1 = 1 if i < 101

Arrays

- Access large amounts of similar data
- **Index**: access each element
- **Size**: number of elements

Size (a) = 6 [0-5]

a[5]
a[4]
a[3]
a[2]
a[1]
a[0]

ELSEVIER

Arrays

- 5-element array
- **Base address** = 0x12348000 (address of first element, `array[0]`)
- First step in accessing an array: load base address into a register

Address	Memory
0x12340010	array[4]
0x1234800C	array[3]
0x12348008	array[2]
0x12348004	array[1]
0x12348000	array[0]

0	1	2	3

ELSEVIER

Accessing Arrays

```
// C Code
   int array[5];
   array[0] = array[0] * 2;
   array[1] = array[1] * 2;
```

Address	Data
0x12340010	array[4]
0x1234800C	array[3]
0x12348008	array[2]
0x12348004	array[1]
0x12348000	array[0]

```
# MIPS assembly code
# $s0 = array base address
  lui  $s0, 0x1234          # 0x1234 in upper half of $s0
  ori  $s0, $s0, 0x8000     # 0x8000 in lower half of $s0

  lw   $t1, 0($s0)          # $t1 = array[0]
  sll  $t1, $t1, 1          # $t1 = $t1 * 2
  sw   $t1, 0($s0)          # array[0] = $t1

  lw   $t1, 4($s0)          # $t1 = array[1]
  sll  $t1, $t1, 1          # $t1 = $t1 * 2
  sw   $t1, 4($s0)          # array[1] = $t1
```

Arrays using For Loops

```
// C Code
   int array[1000];
   int i;

   for (i=0; i < 1000; i = i + 1)
       array[i] = array[i] * 8;

# MIPS assembly code
# $s0 = array base address, $s1 = i
```

COMPUTER ARCHITECTURE

ELSEVIER

Arrays Using For Loops

```
# MIPS assembly code
# $s0 = array base address: 0x23B8F000, $s1 = i
# initialization code
  lui  $s0, 0x23B8         # $s0 = 0x23B80000
  ori  $s0, $s0, 0xF000    # $s0 = 0x23B8F000
  addi $s1, $0, 0          # i = 0
  addi $t2, $0, 1000       # $t2 = 1000

loop:
  slt  $t0, $s1, $t2       # i < 1000?
  beq  $t0, $0, done       # if not then done
  sll  $t0, $s1, 2         # $t0 = i * 4 (byte offset)
  add  $t0, $t0, $s0       # address of array[i]
  lw   $t1, 0($t0)         # $t1 = array[i]
  sll  $t1, $t1, 3         # $t1 = array[i] * 8
  sw   $t1, 0($t0)         # array[i] = array[i] * 8
  addi $s1, $s1, 1         # i = i + 1
  j    loop                # repeat
done:
```


ELSEVIER

ASCII Code

- *American Standard Code for Information Interchange*
- Each text character has unique byte value
  - For example, S = 0x53, a = 0x61, A = 0x41
  - Lower-case and upper-case differ by 0x20 (32)

ELSEVIER

Cast of Characters

#	Char	#	Char	#	Char	#	Char	#	Char	#	Char
20	space	30	0	40	@	50	P	60	`	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	'	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o		

© Digital Design and Computer Architecture, 2nd Edition, 2012

ELSEVIER

Ada Lovelace, 1815-1852

- Wrote the first computer program
- Her program calculated the Bernoulli numbers on Charles Babbage's Analytical Engine
- She was the daughter of the poet Lord Byron

ARCHITECTURE

Summary

- Review R/I/J Type
- R-Type (logical / Shift)
- I-Type (logical)
- J-Type (conditional / non-conditional)
- High level language [C] → Assembly

http://rivoire.cs.sonoma.edu/cs351/wemips/

WeMips: Online Mips Emulator

Not secure | rivoire.cs.sonoma.edu/cs351/wemips/

Line: 1 Go! Show/Hide Demos

User Guide | Unit Tests | Docs

```
1  # Not sure what to do now? Enter your mips code here
2  # and hit step (to run one line at a time) or
3  # hit run (to run them all at once)
4
5  # Keep an eye on the register and stack tracker
6  # to see what changes are being made to them.
7
8  # If you want to preload the stack or some registers
9  # with data, you can click on them to change edit them.
10
11 # If you would like more infomation, check out the user
12 # guide linked in the menu.
13
14
15 ADDI $s0, $zero, 10
16 ADDI $s1, $zero, 9
17 SB $s0, -10($sp)
18 SB $s1, -9($sp)
19 LB $s2, -9($sp)
20 ADDI $sp, $sp, -1
21 ADDI $sp, $sp, -2
22 ADDI $sp, $sp, -3
23 ADDI $sp, $sp, -5
24 ADDI $sp, $sp, -9
25 ADDI $sp, $sp, -14
26 ADDI $sp, $sp, -23
27 ADDI $sp, $sp, -37
28 LB $s3, 0($sp)
```

$0 → $zero

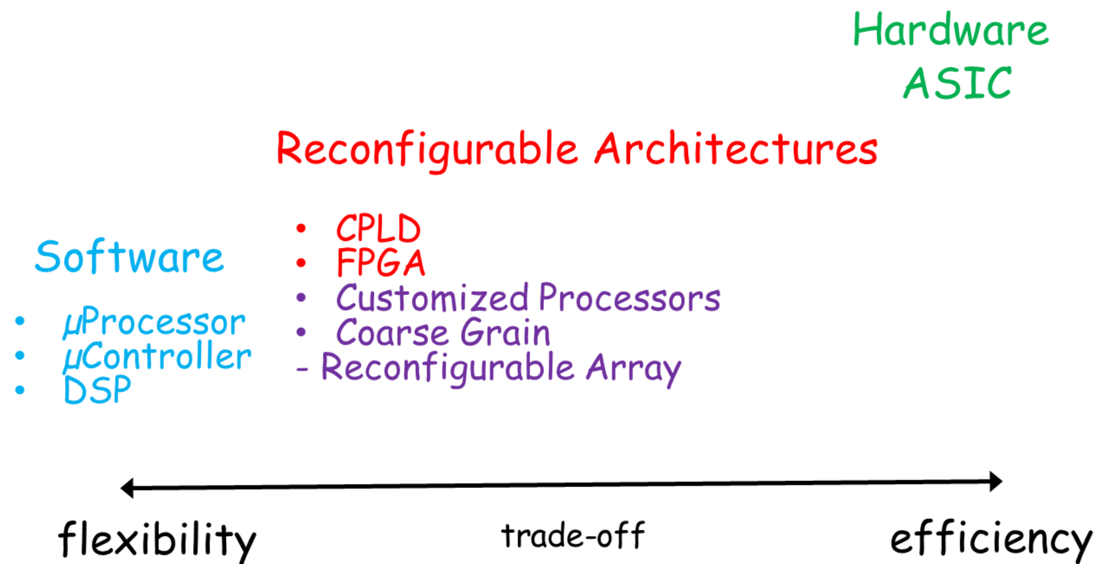
Step Run ☑ Enable auto switching

S T A V Stack Log

s0:	727
s1:	810
s2:	965
s3:	47
s4:	729
s5:	551
s6:	4
s7:	449

Final Project (امتحان العملي)

COMPUTER ARCHITECTURE



ELSEVIER