ISA (Instruction Set Architecture) Part II

Ergun Simsek

CMSC 411 | Lecture 6 | Fall 2022

1

Brief of last class

- von Neumann model of a computer
- Introduction to ISA (Instruction Set Architecture)

Today

- MIPS Instructions
- How to use the MIPS Reference Card (Green Card)

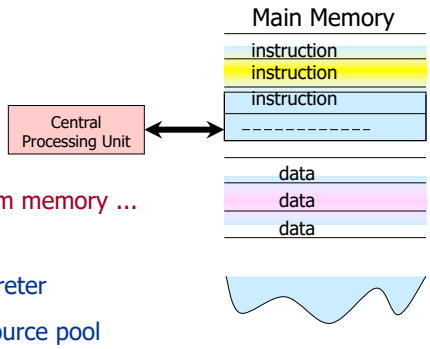
2

The Stored-Program Computer

The von Neumann model:

- Instructions and Data stored in a common memory ("main memory")
- Sequential semantics: All instructions execute sequentially (or at least appear sequential to the programmer)

Key idea: Memory holds not only data, but *coded instructions* that make up a *program*.



- CPU fetches and executes instructions from memory ...
  - The CPU is a hardware interpreter
  - Program IS simply data for this interpreter
  - Main memory: Single expandable resource pool
    - constrains both data and program size
    - don't need to make separate decisions of how large of a program or data memory to buy

CMSC 411 – Lecture 6 3

3

Register File

Register name	Number	Usage
$zero	0	constant 0
$at	1	reserved for assembler
$v0	2	expression evaluation and results of a function
$v1	3	expression evaluation and results of a function
$a0	4	argument 1
$a1	5	argument 2
$a2	6	argument 3
$a3	7	argument 4
$t0	8	temporary (not preserved across call)
$t1	9	temporary (not preserved across call)
$t2	10	temporary (not preserved across call)
$t3	11	temporary (not preserved across call)
$t4	12	temporary (not preserved across call)
$t5	13	temporary (not preserved across call)
$t6	14	temporary (not preserved across call)
$t7	15	temporary (not preserved across call)
$s0	16	saved temporary (preserved across call)
$s1	17	saved temporary (preserved across call)
$s2	18	saved temporary (preserved across call)
$s3	19	saved temporary (preserved across call)
$s4	20	saved temporary (preserved across call)
$s5	21	saved temporary (preserved across call)
$s6	22	saved temporary (preserved across call)
$s7	23	saved temporary (preserved across call)
$t8	24	temporary (not preserved across call)
$t9	25	temporary (not preserved across call)
$k0	26	reserved for OS kernel
$k1	27	reserved for OS kernel
$gp	28	pointer to global area
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address (used by function call)

00000 → $zero

10110 → $s6

11111 → $ra

CMSC 411 – Lecture 6 4

4

Recap: MIPS Instruction Formats

All MIPS instructions fit into a single 32-bit word

Every instruction includes various "fields":

- a 6-bit operation or "OPCODE"
  - specifies which operation to execute (fewer than 64)
- We either have 3 or 2 5-bit OPERAND fields which specify a register (one of 32) as source/destination
- If we don't have three operand fields, then it means we have an "immediate"
  - sometimes treated as signed values, sometimes unsigned

Name	Bit Fields					
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
R-Format	op	rs	rt	rd	shmt	funct
I-format	op	rs	rt	**address/immediate (16)**		
J-format	op	**target address (26)**				

NO FUNCT CODE FOR I- and J- insts.

5

MIPS Reference Data (Green Card)

CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION (in Verilog)	OPCODE / FUNCT (Hex)
Add	add	R	R[rd] = R[rs] + R[rt]	(1) 0 / 20_{hex}
Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm	(1,2) 8_{hex}

Ins	Opcode	Funct	Type
add	000000	100000	R
addi	001000		I

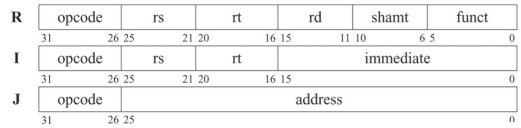
(1) May cause overflow exception
(2) SignExtImm = { 16{immediate[15]}, immediate }
(3) ZeroExtImm = { 16{1b'0}, immediate }
(4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }
(5) JumpAddr = { PC+4[31:28], address, 2'b0 }
(6) Operands considered unsigned numbers (vs. 2's comp.)
(7) Atomic test&set pair; R[rt] = 1 if pair atomic, 0 if not atomic

Const is 16-bit two's comp. sign-extended to 32 bits

addiu

Const is 16-bit two's comp. sign-extended to 32 bits
No overflow trap.

BASIC INSTRUCTION FORMATS



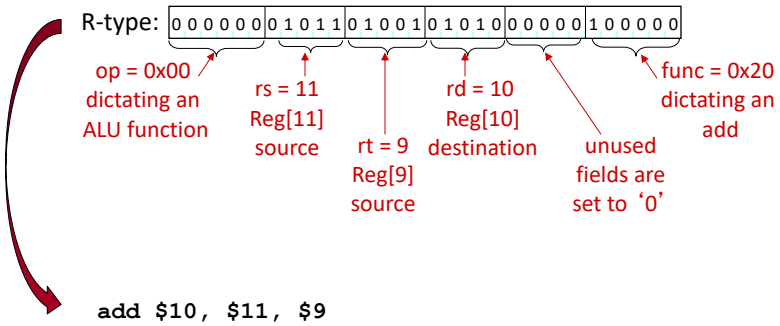
Get familiar with bit numbers!

6

6

MIPS ALU Operations

Sample coded operation: ADD instruction




```
add $10, $11, $9
```

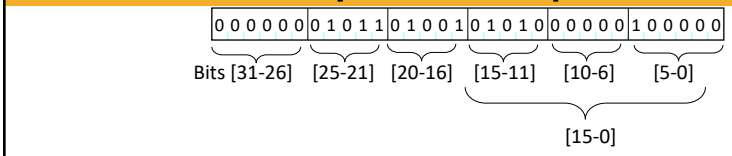
Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three register operands
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Three register operands
	add immediate	addi $s1,$s2,20	$s1 = $s2 + 20	Used to add constants

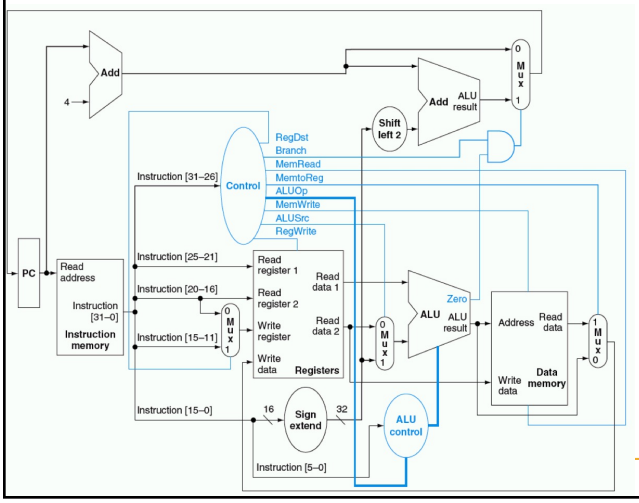
7

7

MIPS Since Cycle Datapath

```
add rd,  rs,  rt
add $10, $11, $9
```





8

8

Function Code Examples

Instruction		Function	Instruction		Function
add	rd, rs, rt	100000	srlv	rd, rt, rs	000110
addu	rd, rs, rt	100001	sub	rd, rs, rt	100010
and	rd, rs, rt	100100	subu	rd, rs, rt	100011
break		001101	syscall		001100
div	rs, rt	011010	xor	rd, rs, rt	100110
divu	rs, rt	011011			
jalr	rd, rs	001001			
jr	rs	001000			
mfhi	rd	010000			
mflo	rd	010010			
mthi	rs	010001			
mtlo	rs	010011			
mult	rs, rt	011000			
multu	rs, rt	011001			
nor	rd, rs, rt	100111			
or	rd, rs, rt	100101			
sll	rd, rt, sa	000000			
sllv	rd, rt, rs	000100			
slt	rd, rs, rt	101010			
sltu	rd, rs, rt	101011			
sra	rd, rt, sa	000011			
srav	rd, rt, rs	000111			
srl	rd, rt, sa	000010			

9

9

Shift operations

Shifting is a common operation

- applied to groups of bits
- used for alignment
- used for "short cut" arithmetic operations
  - X << 1 is often the same as 2*X
  - X >> 1 can be the same as X/2

For example:

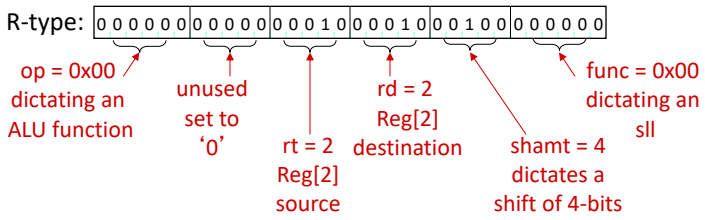
- $X = 20_{10} = 00010100_2$
- Left Shift:
  - $(X << 1) = 00101000_2 = 40_{10}$
- Right Shift:
  - $(X >> 1) = 00001010_2 = 10_{10}$
- Signed Right Shift requires inserting MSB, which is not always 0
  - $(-X >>> 1) = (11101100_2 >>> 1) = 11110110_2 = -10_{10}$

CMSC 411 – Lecture 6

10

MIPS Shift Operations

Sample coded operation: SHIFT LOGICAL LEFT instruction



Assembly: `sll $2, $2, 4`

`sll rd, rt, shamt:`

Reg[rd] = Reg[rt] << shamt

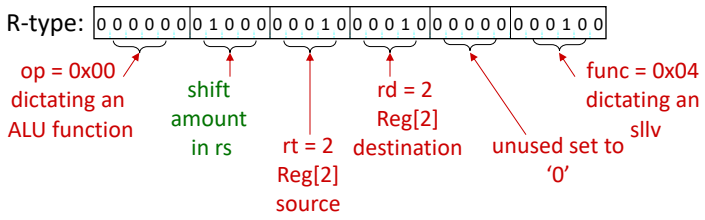
"Shift the contents of *rt* to the left by *shamt*; store the result in *rd*"

11

11

MIPS Shift Operations

Sample coded operation: SLLV (SLL Variable)



Different flavor:
Shift amount is not in instruction, but in a register

Assembly: `sllv $2, $2, $4`

`sllv rd, rt, rs:`

Reg[rd] = Reg[rt] << Reg[rs]

"Shift the contents of *rt* left by the contents of *rs*; store the result in *rd*"

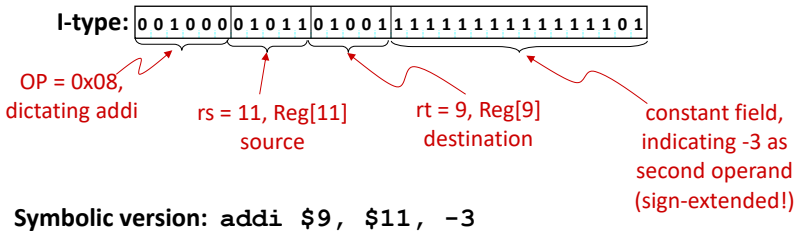
12

12

MIPS ALU Operations with Immediate

FACT: In a C compiler (gcc), 52% of ALU operations use a constant!

addi instruction: adds register contents, signed-constant:



Symbolic version: `addi $9, $11, -3`

`addi rt, rs, imm`:

Reg[rt] = Reg[rs] + sxt(imm)

"Add the contents of rs to const; store result in rt"

Similar instructions for other ALU operations:

arithmetic: addi, addiu
compare: slti, sltiu
logical: andi, ori, xori, lui

Immediate values are sign-extended for arithmetic and compare operations, but not for logical operations.

13

13

Working with Constants

- Examples
  - add 2000 to register $5
    ```
    addi $5, $5, 2000
    ```
  - subtract 60 from register $5
    ```
    addi $5, $5, -60
    ```
    – … no `subi` instruction!
  - put the number 1234 in $10
    ```
    addi $10, $0, 1234
    ```
  - logically AND $5 with 0x8723 and put the result in $7
    ```
    andi $7, $5, 0x8723
    ```
- But…
  - these constants are limited to 16 bits only!
    – Range is [-32768…32767] if signed, or [0…65535] if unsigned

$2^{15} = 32768$

CMSC 411 – Lecture 6 14

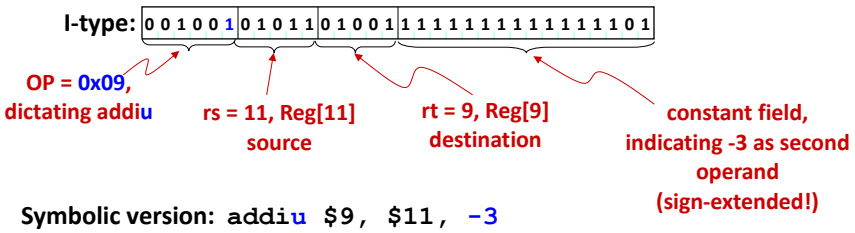
14

Beware ADDIU: "add immediate ~~unsigned~~"

addiu: supposedly "add immediate ~~unsigned~~"

BUT IS A MISNOMER! Actually sign-extends the immediate.

The difference between addi & addiu is that **addiu** doesn't check for overflows



Symbolic version: `addiu $9, $11, -3`

`addiu rt, rs, imm:`

Reg[rt] = Reg[rs] + sign-ext(imm)

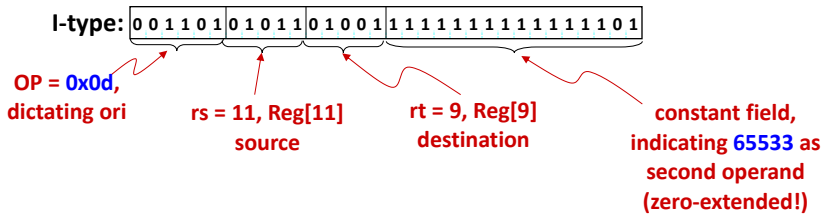
"Add the contents of rs to const; store result in rt"

15

15

ORI: Unsigned Constants

ori instruction: bitwise OR's register to unsigned-constant:



Symbolic version: `ori $9, $11, 65533`

`ori rt, rs, imm:`

Reg[rt] = Reg[rs] | zero-ext(imm)

"OR the contents of rs to const; store result in rt"

The imm is 0-padded into a 32-bit unsigned (+ve) number:

00000000000000001111111111111101

Also: All logical operations are always "unsigned", so always zero-extended:
ori, andi, xori

16

How About Larger Constants?

Problem: How do we work with bigger constants?

- Example: Put the 32-bit value 0x5678ABCD in $5
- CLASS: How will you do it?

One Solution:

- put the upper half (0x5678) into $5
- then shift it left by 16 positions (0x5678 0000)
- now "add" the lower half to it (0x5678 0000 + 0xABCD)

```
addi $5, $0, 0x5678
sll  $5, $5, 16
addi $5, $5, 0xABCD
```

One minor problem with this:

- 2nd **addi** can mess up by treating the constants are signed
- use **ori** instead

How About Larger Constants?

Observation: This sequence is very common!

- so, a special instruction was introduced to make it shorter
- the first two (**addi + sll**) combo is performed by

lui

"load upper immediate"

➢ puts the *16-bit* immediate into the *upper* half of a register

- Example: Put the 32-bit value 0x5678ABCD in $5

```
lui $5, 0x5678
ori $5, $5, 0xABCD
```

0101011001111000	0000000000000000
0000000000000000	1010101111001101
0101011001111000	1010101111001101

Reminder: In MIPS, Logical Immediate instructions (ANDI, ORI, XORI) do not sign-extend their constant operand

MIPS Operations

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three register operands
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Three register operands
	add immediate	addi $s1,$s2,20	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	lw $s1,20($s2)	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	sw $s1,20($s2)	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	lh $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	lhu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	sh $s1,20($s2)	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	lb $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	lbu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	sb $s1,20($s2)	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	ll $s1,20($s2)	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	sc $s1,20($s2)	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	lui $s1,20	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	and $s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	nor $s1,$s2,$s3	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi $s1,$s2,20	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	ori $s1,$s2,20	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	beq $s1,$s2,25	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne $s1,$s2,25	if ($s1!= $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for beq, bne
	set on less than unsigned	sltu $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	slti $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	sltiu $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr $ra	go to $ra	For switch, procedure return
	jump and link	jal 2500	$ra = PC + 4; go to 10000	For procedure call

19

19

Exercise

Compute the following expression with MIPS Assembly

$$f = (g + h) – (i + j)$$

where variables f, g, h, i, and j are assigned to registers $16, $17, $18, $19, and $20 respectively

Register name	Number	Usage
$zero	0	constant 0
$at	1	reserved for assembler
$v0	2	expression evaluation and results of a function
$v1	3	expression evaluation and results of a function
$a0	4	argument 1
$a1	5	argument 2
$a2	6	argument 3
$a3	7	argument 4
$t0	8	temporary (not preserved across call)
$t1	9	temporary (not preserved across call)
$t2	10	temporary (not preserved across call)
$t3	11	temporary (not preserved across call)
$t4	12	temporary (not preserved across call)
$t5	13	temporary (not preserved across call)
$t6	14	temporary (not preserved across call)
$t7	15	temporary (not preserved across call)
$s0	16	saved temporary (preserved across call)
$s1	17	saved temporary (preserved across call)
$s2	18	saved temporary (preserved across call)
$s3	19	saved temporary (preserved across call)
$s4	20	saved temporary (preserved across call)
$s5	21	saved temporary (preserved across call)
$s6	22	saved temporary (preserved across call)
$s7	23	saved temporary (preserved across call)
$t8	24	temporary (not preserved across call)
$t9	25	temporary (not preserved across call)
$k0	26	reserved for OS kernel
$k1	27	reserved for OS kernel
$gp	28	pointer to global area
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address (used by function call)

```
add $8,$17,$18
add $9,$19,$20
sub $16,$8,$9
```

CMSC 411 – Lecture 6 20

20