Lecture 3: MIPS Instruction Set

Machine Language

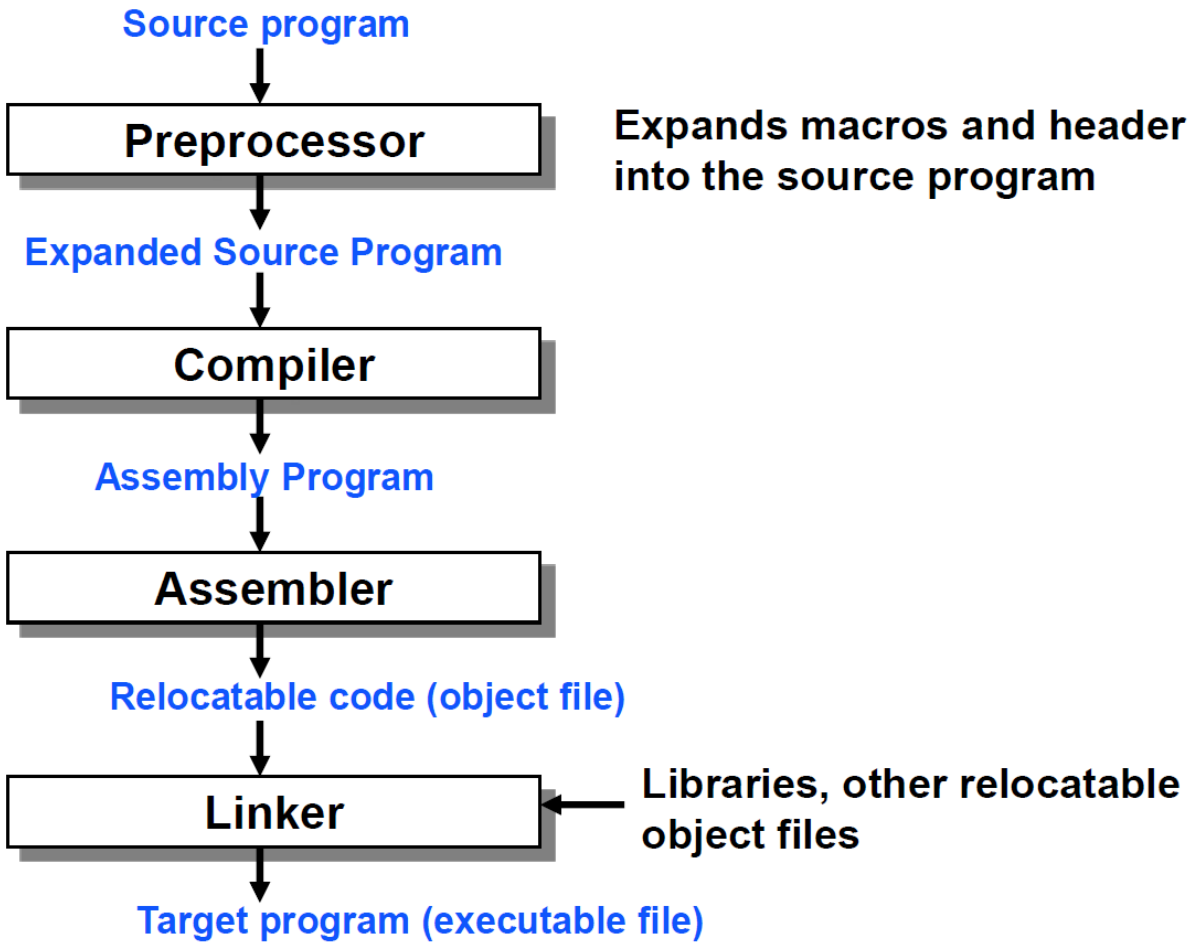
- **Programming language**
  - High-level programming languages
    - Procedural (imperative) languages: C, PASCAL, FORTRAN
    - Object-oriented languages: C++, Objective-C, Java
    - Functional (declarative) languages: Lisp, Scheme
    - Multiple paradigms (hybrid): Python, C#
  - Assembly programming languages: symbolic machine languages
  - Machine languages: binary (1's and 0's), IA32, IA64, ARM, MIPS, PowerPC
- **Translator**
  - Compiler: translate a high-level language program into a machine language program
  - Assembler: translate an assembly language program into a machine language program
    - A part of a compiler
  - Interpreter: translate and execute programs directly
    - JVM(Java virtual machine): translate/execute Java bytecode to native machine instructions
      - cf. Java compiler translate Java program to Java bytecode

Compiler

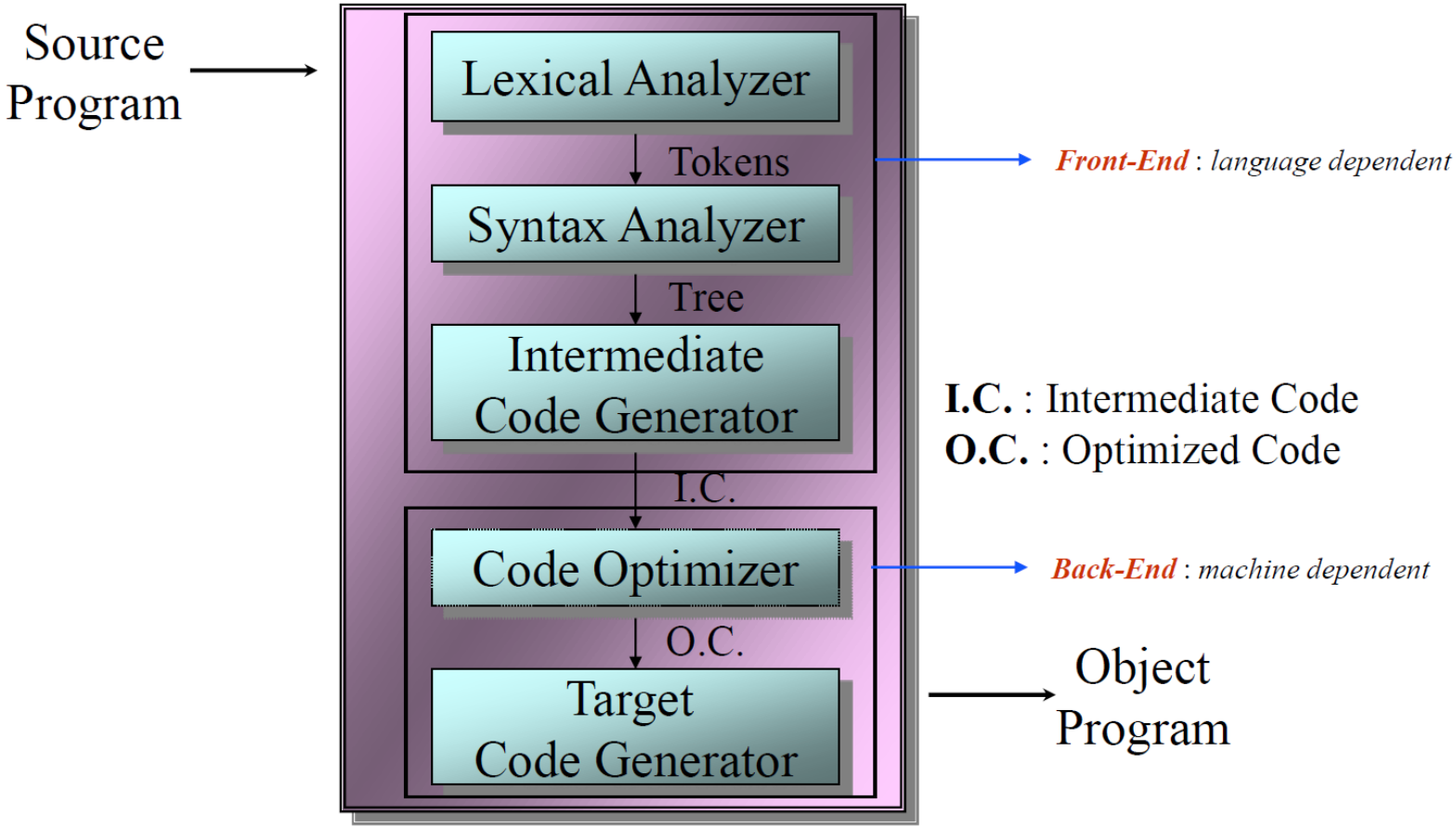
- **Compiler**
  - A program that translates a source program (written in language A) into an equivalent target program (written in language B)

  - Source program
    - Usually written in high-level programming languages (called ***source language***) such as C, C++, Java, FORTRAN, Python
  - Target program
    - Usually written in machine languages (called ***target language***) such as x86, Alpha, MIPS, SPARC, or ARM instructions
- **What qualities do you want in a compiler?**
  - Generate correct code
  - Target code runs fast
  - Compiler runs fast
  - Support for separate compilation, good diagnostics for errors

Compilation Process



Compiler Phases



Machine State

- **ISA defines machine states and instructions**
- **Registers**
  - CPU internal storage to *store data* fetched from memory
    - Can be read or written in a single cycle
    - Arithmetic and logic operations are usually performed on registers
    - MIPS ISA has 32 32-bit registers: Each register consists of 32 flip-flops
  - Top level of the memory hierarchy
    - Registers <-> caches <-> memory <-> hard disk
      - Registers are visible to programmers and maintained by programmers
      - Caches are invisible to programmers and maintained by HW
- **Memory**
  - A large, single dimensional array, starting at address 0
    - To access a data item in memory, an instruction must supply an address.
  - Store programs (which contains both instructions and data)
  - To transfer data, use load (memory to register) and store (register to memory) instructions

Data Size & Alignment

- **Data size**
  - *Word* : the default data size for computation
    - 32b for 32b ISA, 64b for 64b ISA
  - *Double word*: 64b data, *Half word*: 16b data, *Byte*: 8b data
    - Load/store instructions can designate data sizes transferred: ldw, lddw, ldhw, ldb
- **Byte addressability**
  - Each byte has an address
- **Alignment**
  - Objects must start at addresses that are multiple of their size

Object addressed	Aligned addresses	Misaligned addresses
Byte	0, 1, 2, 3, 4, 5, 6, 7	Never
Half Word	0, 2, 4, 6	1, 3, 5, 7
Word	0, 4	1, 2, 3, 5, 6, 7
Double Word	0	1, 2, 3, 4, 5, 6, 7

Machine Instruction

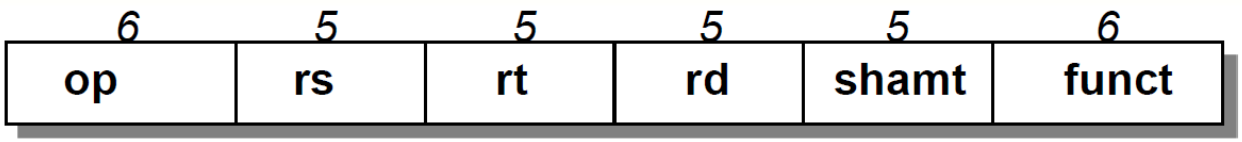
- **Opcode : specifies the operation to be performed**
  - EX) ADD, MULT, LOAD, STORE, JUMP
- **Operands : specifies the location of data**
  - Source operands (input data)
  - Destination operands (output data)
  - The location can be
    - Memory operand specified by a memory address : EX) 8(R2), x1004F
    - Register operand specified by a register number : R1

MIPS Instruction Types

- **Arithmetic and logic instructions**
  - Performs actual computation on operands
  - EX) ADD, MULT, XOR, SHIFT, FDIVIDE, FADD
- **Data transfer instructions (memory instructions)**
  - Move data from memory to registers (LOAD) or vice versa (STORE)
  - Input/Output instructions are usually implemented by memory instructions (***memory-mapped IO***)
    - IO devices are mapped to memory address space
- **Control transfer instructions (branch instructions)**
  - Change the program control flow (i.e. change the PC)
    - Determine the address of the next instruction to be fetched
      - **Program Counter (PC)**: A special register that holds the address of the next instruction to execute
  - Unconditional jumps and conditional branches
  - Direct branches and indirect branches
  - EX) JUMP, CALL, RETURN, BEQ

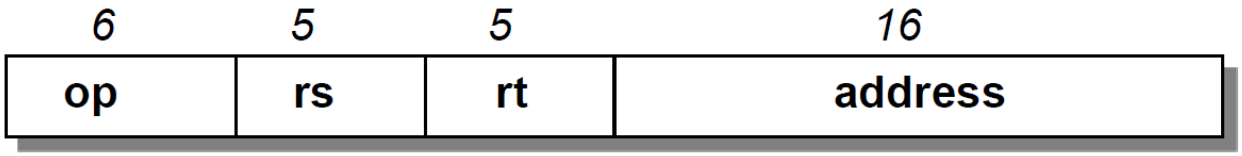
MIPS Instruction Format

R-type



- Op: Opcode, basic operation of the instruction
- Rs: 1st source register
- Rt: 2nd source register
- Rd: destination register
- shamt: shift amount
- funct: function code, the specific variant of the opcode
- Used for arithmetic/logic instructions

I-type



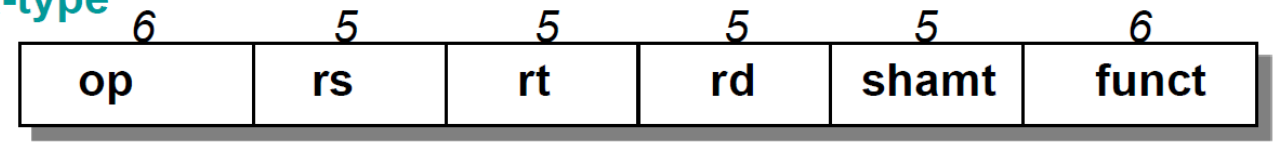
- Rs: ***base register***
- Address: +/- 2^{15} bytes ***offset*** (or also called ***displacement***)
- Used for loads/stores and conditional branches

MIPS Addressing Modes

- **Register addressing**
  - Address is in a register
  - jr $ra (***indirect branches***)
- **Base addressing**
  - Address is the sum of a register (***base register***) and a constant (***offset***)
  - ldw $s0, 100($s1)
- **Immediate addressing**
  - For constant operand (***immediate*** operand)
  - add $t1, $t2, 3
- **PC-relative addressing**
  - Address is the sum of PC and a constant (***offset***)
  - beq $s0, $s1, L1
- **Pseudodirect addressing**
  - Address is the 26 bit offset concatenated with the upper bits of PC
  - j L1

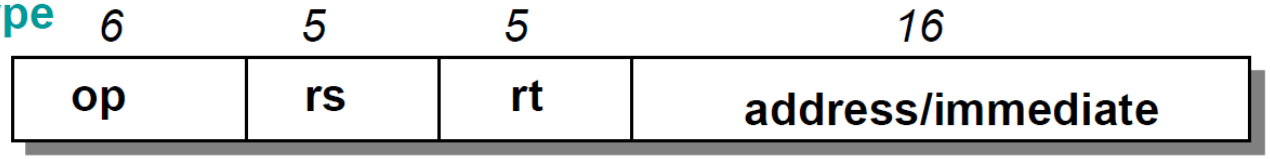
MIPS Instruction Format

R-type



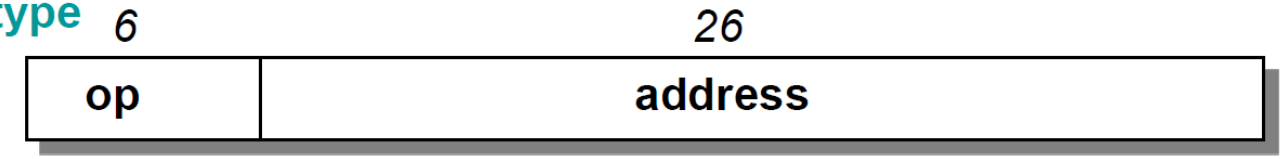
- Arithmetic instructions

I-type



- Data transfer, conditional branch, immediate format instructions

J-type



- Jump instructions

MIPS Instruction Example: R-format

- MIPS Instruction:
  - add $8,$9,$10

Decimal number per field representation:

0	9	10	8	0	32

Binary number per field representation:

000000	01001	01010	01000	00000	100000

hex

hex representation: $012A\ 4020_{hex}$

decimal representation: $19,546,144_{ten}$

- Called a Machine Language Instruction

MIPS Instruction Opcode Table

op(31:26)								
28–26 31–29	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	R-format	Bltz/gez	jump	jump & link	branch eq	branch ne	blez	bgtz
1(001)	add immediate	addiu	set less than imm.	set less than imm. unsigned	andi	ori	xori	load upper immediate
2(010)	TLB	FlPt						
3(011)								
4(100)	load byte	load half	lwl	load word	load byte unsigned	load half unsigned	lwr	
5(101)	store byte	store half	swl	store word			swr	
6(110)	load linked word	lwc1						
7(111)	store cond. word	swc1						

op(31:26)=010000 (TLB), rs(25:21)								
23–21 25–24	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(00)	mfc0		cfc0		mtc0		ctc0	
1(01)								
2(10)								
3(11)								

op(31:26)=000000 (R-format), funct(5:0)								
2–0 5–3	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	shift left logical		shift right logical	sra	sllv		srlv	srav
1(001)	jump register	jalr			syscall	break		
2(010)	mfhi	mthi	mflo	mtlo				
3(011)	mult	multu	div	divu				
4(100)	add	addu	subtract	subu	and	or	xor	not or (nor)
5(101)			set l.t.	set l.t. unsigned				
6(110)								
7(111)								

MIPS Instruction Example

MIPS machine language

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add $s1,$s2,$s3
sub	R	0	18	19	17	0	34	sub $s1,$s2,$s3
addi	I	8	18	17	100			addi $s1,$s2,100
lw	I	35	18	17	100			lw $s1,100($s2)
sw	I	43	18	17	100			sw $s1,100($s2)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

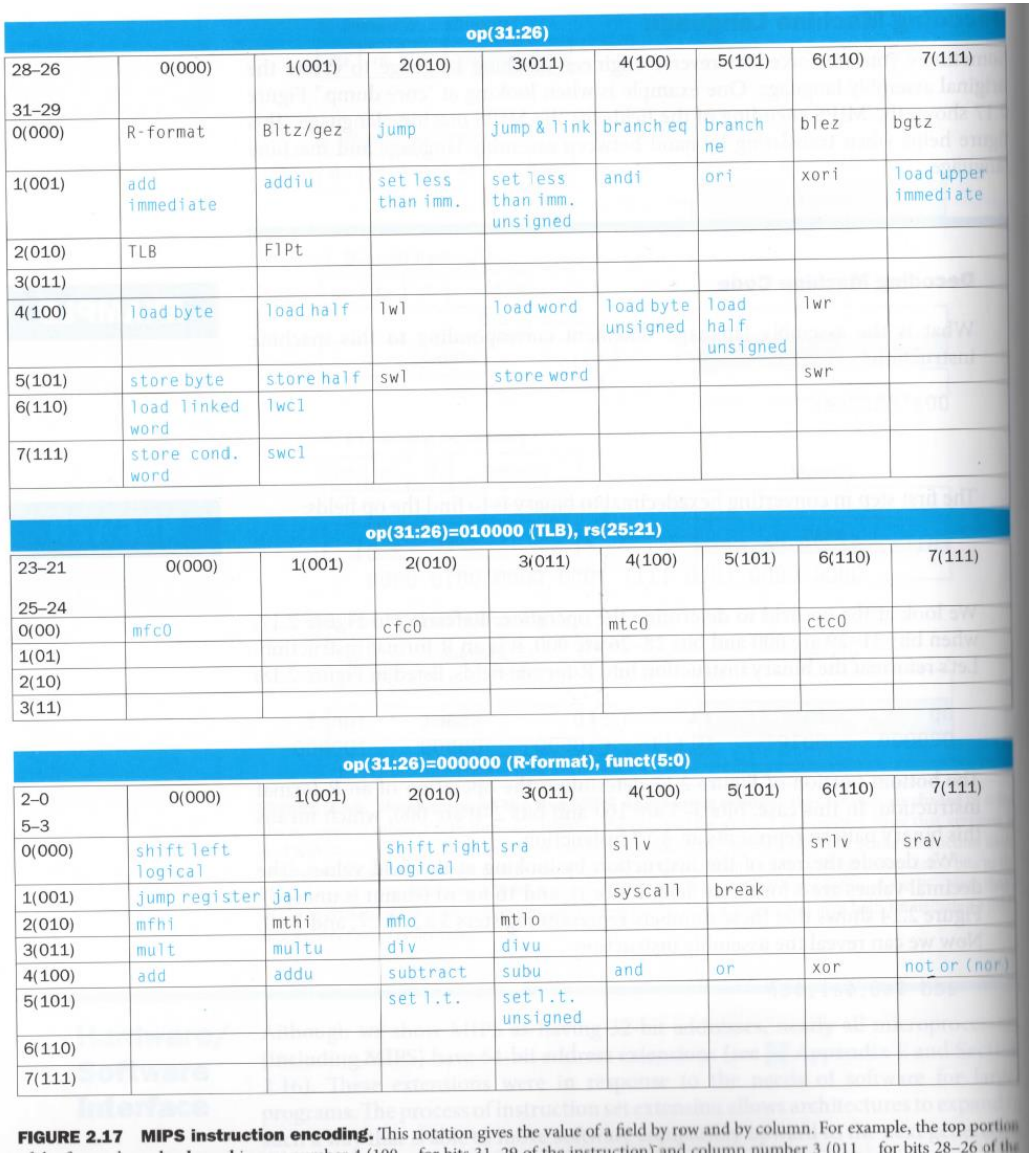
MIPS Instruction Example

op(31:26)								
28–26 31–29	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	R-format	Bltz/gez	jump	jump & link	branch eq	branch ne	blez	bgtz
1(001)	add immediate	addiu	set less than imm.	set less than imm. unsigned	andi	ori	xori	load upper immediate
2(010)	TLB	FlPt						
3(011)								
4(100)	load byte	load half	lwl	load word	load byte unsigned	load half unsigned	lwr	
5(101)	store byte	store half	swl	store word			swr	
6(110)	load linked word	lwc1						
7(111)	store cond. word	swc1						

op(31:26)=010000 (TLB), rs(25:21)								
23–21 25–24	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(00)	mfc0		cfc0		mtc0		ctc0	
1(01)								
2(10)								
3(11)								

op(31:26)=000000 (R-format), funct(5:0)								
2–0 5–3	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	shift left logical		shift right logical	sra	sllv		srlv	srav
1(001)	jump register	jalr			syscall	break		
2(010)	mfhi	mthi	mflo	mtlo				
3(011)	mult	multu	div	divu				
4(100)	add	addu	subtract	subu	and	or	xor	not or (nor)
5(101)			set l.t.	set l.t. unsigned				
6(110)								
7(111)								

MIPS Instruction Example



op(31:26)								
28–26 31–29	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	R-format	Bltz/gez	jump	jump & link	branch eq	branch ne	blez	bgtz
1(001)	add immediate	addiu	set less than imm.	set less than imm. unsigned	andi	ori	xori	load upper immediate
2(010)	TLB	FlPt						
3(011)								
4(100)	load byte	load half	lwl	load word	load byte unsigned	load half unsigned	lwr	
5(101)	store byte	store half	swl	store word			swr	
6(110)	load linked word	lwc1						
7(111)	store cond. word	swc1						

op(31:26)=010000 (TLB), rs(25:21)								
23–21 25–24	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(00)	mfc0		cfc0		mtc0		ctc0	
1(01)								
2(10)								
3(11)								

op(31:26)=000000 (R-format), funct(5:0)								
2–0 5–3	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	shift left logical		shift right logical	sra	sllv		srlv	srav
1(001)	jump register	jalr			syscall	break		
2(010)	mfhi	mthi	mflo	mtlo				
3(011)	mult	multu	div	divu				
4(100)	add	addu	subtract	subu	and	or	xor	not or (nor)
5(101)			set l.t.	set l.t. unsigned				
6(110)								
7(111)								

FIGURE 2.17 MIPS instruction encoding. This notation gives the value of a field by row and by column. For example, the top portion

MIPS Instruction Example

MIPS operands

Name	Example	Comments
32 registers	$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at	Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register $zero always equals 0, and register $at is reserved by the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

MIPS assembly language

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

A Basic MIPS Instruction

C code: a = b + c ;

Assembly code: (human-friendly machine instructions)

```
add   a, b, c      #  a is the sum of b and c
```

Machine code: (hardware-friendly machine instructions)

```
00000010001100100100000000100000
```

Translate the following C code into assembly code:

a = b + c + d + e;

Example

C code a = b + c + d + e;

translates into the following assembly code:

```
add  a, b, c          add  a, b, c
add  a, a, d    or    add  f, d, e
add  a, a, e          add  a, a, f
```

- Instructions are simple: fixed number of operands (unlike C)
- A single line of C code is converted into multiple lines of assembly code
- Some sequences are better than others… the second sequence needs one more (temporary) variable f

Example

Convert to assembly:

C code: d[3] = d[2] + a;

Example

Convert to assembly:

C code: d[3] = d[2] + a;

Assembly: # addi instructions as before

```
lw   $t0, 8($s4)   #  d[2] is brought into $t0
lw   $t1, 0($s1)   #   a  is brought into $t1
add  $t0, $t0, $t1  #  the sum is in $t0
sw   $t0, 12($s4)  #  $t0 is stored into d[3]
```

Assembly version of the code continues to expand!

Another Version

Convert to assembly:

C code: d[3] = d[2] + a;

Assembly:

```
lw    $t0, 20($gp)    #  d[2] is brought into $t0
lw    $t1, 0($gp)     #   a  is brought into $t1
add   $t0, $t0, $t1   #  the sum is in $t0
sw    $t0, 24($gp)    #  $t0 is stored into d[3]
```