EEC170
Computer Architecture

Lecture 2
Addressing Modes and MIPS ISA

October 10, 2005

Soheil Ghiasi
Electrical and Computer Engineering
University of California, Davis

Announcements!

- Slides are online
  - Password protected:
    - Username: EEC289Q
    - Password: FPGAsrcool!
- TA office hours are Mondays right after class
  - M 4-6pm
  - 2101 Kemper Hall

Key Design Principles

1. Simplicity favors regularity.
2. Smaller is faster.
3. Make the common case fast.
4. Good design demands good compromises.

Recap: Basic ISA Classes

- Memory to Memory:
  - 2 address — add A B — mem[A] ← mem[A] + mem[B]
  - 3 address — add A B C — mem[A] ← mem[B] + mem[C]
- Accumulator:
  - 1 address — add A — acc ← acc + mem[A]
  - 1+x address — addx A — acc ← acc + mem[A + x]
- Stack:
  - 0 address — add — tos ← tos + next
- General Purpose Register:
  - 2 address — add A B — reg[A] ← reg[A] + reg[B]
  - 3 address — add A B C — reg[A] ← reg[B] + reg[C]

Comparison:

Bytes per instruction? Number of Instructions? Cycles per instruction?

Instruction Classes, Format, Addressing Modes

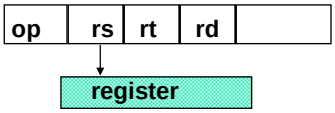
- Instruction Classes
  - Expect new instruction set architectures to use general purpose register
- Instruction Format
  - If code size is most important, use variable length instructions
  - If performance is most important, use fixed length instructions
- Data Addressing Modes
  - Frequent: Displacement, Immediate, Register Indirect
  - Displacement size should be 12 to 16 bits
  - Immediate size should be 8 to 16 bits
- Operand Sizes
  - Support these data sizes and types:
    8-bit, 16-bit, 32-bit, 64-bit integers and
    32-bit and 64-bit IEEE 754 floating point numbers

Addressing Modes

- How do we specify the location of the operands?
- Number of different possible Addressing Modes
  - Register (direct)
  - Immediate
  - Register Indirect
  - Relative
  - PC relative
  - Indexed

Register (Direct)

- Idea: the instruction specifies the register which stores the data



- Number of memory accesses? Zero
- Number of bits in instruction needed to specify operand? $\log_2$ (# registers)
- Example: add R1, R2, R3

Immediate Mode

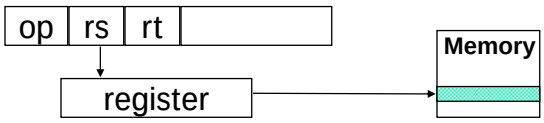
- Idea: the instruction contains the operand data
  - Used for "constant" operands

- Number of memory accesses? Zero
- Number of bits in instruction to specify operand?

  Depends on the precision
- Example: add R1, R2, 5

Register Indirect

- Idea: a register contains the "address" in memory of the operand
  - Basically, use a register as a pointer, instead of using a memory variable

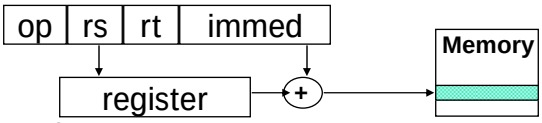


- Number of memory accesses? 1
  - (must have previously loaded the address into the register)
- Number of bits in instruction? $\log_2$ (# registers)
- Example: vs.

la reg3, myvar	la addr1, myvar
add [reg3], 3	add m(addr1), 3

Relative Mode

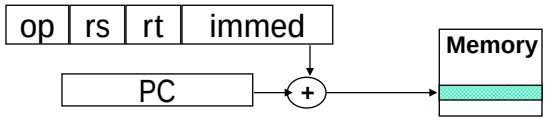
- Idea: a register contains an "address"; add a "small" constant to get the effective address of the operand



- Number of memory addresses? 1
- Number of bits in instruction? $\log_2$(# reg) + precision(immed)
- Example:
  - load_address reg3, myarray
  - add 4[reg3], 3
- Called base or displacement addressing by the book

PC-relative Mode

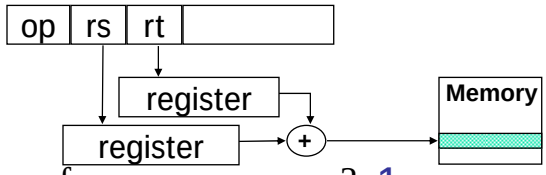
- Modification to relative mode; the base register is implicit = to the program counter (PC)
  - PC contains address of the next instruction to execute
  - So branch to a location some displacement away from next instruction
  - Sole benefit: displacements are smaller than absolute address



- Number of memory accesses? 1
- Number of bits in instruction? precision(immed)

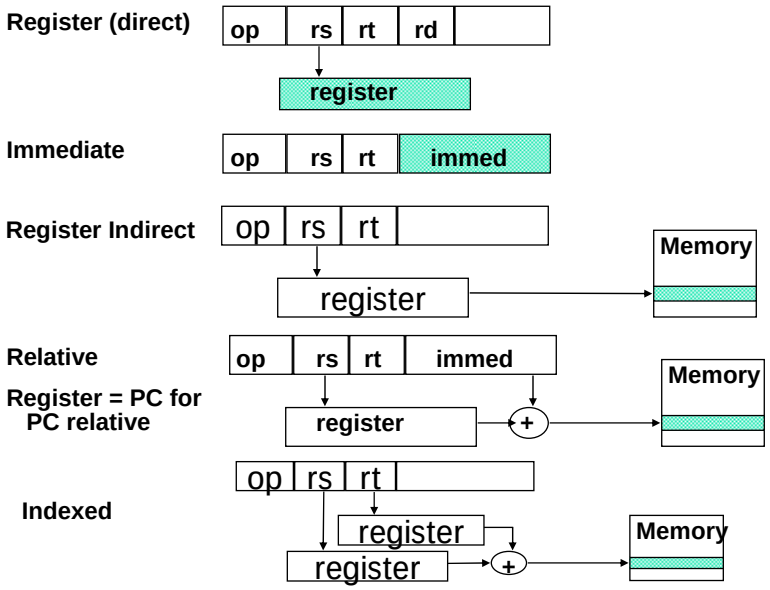
Indexed Mode

- Idea: one register contains an "address", add value in another register to get the effective address of the operand



- Number of memory accesses? 1
- Number of bits in instruction? $2 * \log_2(\#\ reg)$
- Example:
  - load_address reg1, myarray
  - load_address reg2, displacement
  - add [reg1 + reg2], 3 –or– add[reg1][reg2], 3

Summary: Addressing Modes



MIPS I Operation Overview

- Arithmetic/Logical:
  - Add, AddU, Sub, SubU, And, Or, Xor, Nor, SLT, SLTU
  - AddI, AddIU, SLTI, SLTIU, AndI, OrI, XorI
  - SLL, SRL, SRA, SLLV, SRLV, SRAV
- Memory Access:
  - LB, LBU, LH, LHU, LW, LWL,LWR
  - SB, SH, SW, SWL, SWR

MIPS logical instructions

Instruction	*Example*	*Meaning*	*Comment*
and	**and $1,$2,$3**	**$1 = $2 & $3**	**3 reg. operands; Logical AND**
or	**or $1,$2,$3**	**$1 = $2 | $3**	**3 reg. operands; Logical OR**
xor	**xor $1,$2,$3**	**$1 = $2 ^ $3**	**3 reg. operands; Logical XOR**
nor	**nor $1,$2,$3**	**$1 = ~($2 |$3)**	**3 reg. operands; Logical NOR**
and immediate	**andi $1,$2,10**	**$1 = $2 & 10**	**Logical AND reg, constant**
or immediate	**ori $1,$2,10**	**$1 = $2 | 10**	**Logical OR reg, constant**
xor immediate	**xori $1, $2,10**	**$1 = $2 ^ 10**	**Logical XOR reg, constant**
shift left logical	**sll $1,$2,10**	**$1 = $2 << 10**	**Shift left by constant**
shift right logical	**srl $1,$2,10**	**$1 = $2 >> 10**	**Shift right by constant**
shift right arithm.	**sra $1,$2,10**	**$1 = $2 >> 10**	**Shift right (sign extend)**
shift left logical	**sllv $1,$2,$3**	**$1 = $2 << $3**	**Shift left by variable**
shift right logical	**srlv $1,$2, $3**	**$1 = $2 >> $3**	**Shift right by variable**
shift right arithm.	**srav $1,$2, $3**	**$1 = $2 >> $3**	**Shift right arith. by variable**

Q: How multiply by 2^i ? Divide by 2^i ? Mult by 15? Invert?

MIPS Reference Data: CORE INSTRUCTION SET

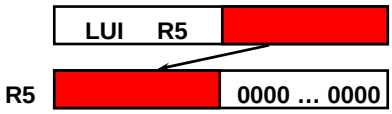
NAME	MNE-MON-IC	FOR-MAT	OPERATION (in Verilog)	OPCODE /FUNCT (hex)
Add	add	R	R[rd] = R[rs] + R[rt] (1)	$0 / 20_{hex}$
Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm (1)(2)	8_{hex}
Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+ BranchAddr (4)	4_{hex}

(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}

MIPS data transfer instructions

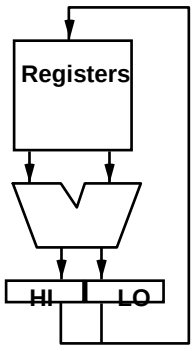
Instruction	*Comment*
sw 500($4), $3	Store word
sh 502($2), $3	Store half
sb 41($3), $2	Store byte
lw $1, 30($2)	Load word
lh $1, 40($3)	Load halfword
lhu $1, 40($3)	Load halfword unsigned
lb $1, 40($3)	Load byte
lbu $1, 40($3)	Load byte unsigned
lui $1, 40	Load Upper Immediate (16 bits shifted left by 16)

Q: Why need lui?



Multiply / Divide

- Start multiply, divide
  - MULT rs, rt
  - MULTU rs, rt
  - DIV rs, rt
  - DIVU rs, rt
- Move result from multiply, divide
  - MFHI rd
  - MFLO rd
- Move to HI or LO
  - MTHI rd
  - MTLO rd



MIPS arithmetic instructions

Instruction	*Example*	*Meaning*	*Comments*
add	add $1,$2,$3	$1 = $2 + $3	3 operands; exception possible
subtract	sub $1,$2,$3	$1 = $2 – $3	3 operands; exception possible
add immediate	addi $1,$2,100	$1 = $2 + 100	+ constant; exception possible
add unsigned	addu $1,$2,$3	$1 = $2 + $3	3 operands; no exceptions
subtract unsigned	subu $1,$2,$3	$1 = $2 – $3	3 operands; no exceptions
add imm. unsign.	addiu $1,$2,100	$1 = $2 + 100	+ constant; no exceptions
multiply	mult $2,$3	Hi, Lo = $2 x $3	64-bit signed product
multiply unsigned	multu$2,$3	Hi, Lo = $2 x $3	64-bit unsigned product
divide	div $2,$3	Lo = $2 ÷ $3,	Lo = quotient, Hi = remainder Hi = $2 mod $3
divide unsigned	divu $2,$3	Lo = $2 ÷ $3,	Unsigned quotient & remainder Hi = $2 mod $3
Move from Hi	mfhi $1	$1 = Hi	Used to get copy of Hi
Move from Lo	mflo $1	$1 = Lo	Used to get copy of Lo

Q: Which add for address arithmetic? Which add for integers?

When does MIPS sign extend?

- When value is sign extended, copy upper bit to full value:

  Examples of sign extending 8 bits to 16 bits:
  0 0001010 ⇒ 00000000 0 0001010
  1 0001100 ⇒ 11111111 1 0001100

- When is an immediate operand sign extended?
  - Arithmetic instructions (add, sub, etc.) always sign extend immediates *even for the unsigned versions of the instructions!*
  - Logical instructions do not sign extend immediates (They are zero extended)
  - Load/Store address computations always sign extend immediates
- Multiply/Divide have no immediate operands however:
  - "unsigned" ⇒ treat operands as unsigned
- The data loaded by the instructions lb and lh are extended as follows ("unsigned" ⇒ don't extend sign):
  - lbu, lhu are zero extended
  - lb, lh are sign extended

MIPS Compare and Branch

- Compare and Branch
  - BEQ rs, rt, offset if R[rs] == R[rt] then PC-relative branch
  - BNE rs, rt, offset <>
- Compare to zero and Branch
  - BLEZ rs, offset if R[rs] <= 0 then PC-relative branch
  - BGTZ rs, offset >
  - BLT <
  - BGEZ >=
  - BLTZAL rs, offset if R[rs] < 0 then branch and link (into R 31)
  - BGEZAL >=!
- Remaining set of compare and branch ops take two instructions
- Almost all comparisons are against zero!

MIPS jump, branch, compare instruction

Instruction	*Example*	*Meaning*
branch on equal	**beq $1,$2,100**	**if ($1 == $2) go to PC+4+100** ***Equal test; PC relative branch***
branch on not eq.	**bne $1,$2,100**	**if ($1!= $2) go to PC+4+100** ***Not equal test; PC relative***
set on less than	**slt $1,$2,$3**	**if ($2 < $3) $1=1; else $1=0** ***Compare less than; 2's comp.***
set less than imm.	**slti $1,$2,100**	**if ($2 < 100) $1=1; else $1=0** ***Compare < constant; 2's comp.***
set less than uns.	**sltu $1,$2,$3**	**if ($2 < $3) $1=1; else $1=0** ***Compare less than; natural numbers***
set l. t. imm. uns.	**sltiu $1,$2,100**	**if ($2 < 100) $1=1; else $1=0** ***Compare < constant; natural numbers***
jump	**j 10000**	**go to 10000** ***Jump to target address***
jump register	**jr $31**	**go to $31** ***For switch, procedure return***
jump and link	**jal 10000**	**$31 = PC + 4; go to 10000** ***For procedure call***

Signed vs. Unsigned Comparison

$1= 0…00 0000 0000 0000 0001$_{two}$

$2= 0…00 0000 0000 0000 0010$_{two}$

$3= 1…11 1111 1111 1111 1111$_{two}$

- After executing these instructions:

```
slt  $4,$2,$1 ;  if ($2 < $1) $4=1; else $4=0
slt  $5,$3,$1 ;  if ($3 < $1) $5=1; else $5=0
sltu $6,$2,$1 ;  if ($2 < $1) $6=1; else $6=0
sltu $7,$3,$1 ;  if ($3 < $1) $7=1; else $7=0
```

- What are values of registers $4 - $7? Why?

$4 = ; $5 = ; $6 = ; $7 = ;

Signed vs. Unsigned Comparison

$1= 0…00 0000 0000 0000 0001$_{two}$

$2= 0…00 0000 0000 0000 0010$_{two}$

$3= 1…11 1111 1111 1111 1111$_{two}$

- After executing these instructions:

```
slt  $4,$2,$1 ;  if ($2 < $1) $4=1; else $4=0
slt  $5,$3,$1 ;  if ($3 < $1) $5=1; else $5=0
sltu $6,$2,$1 ;  if ($2 < $1) $6=1; else $6=0
sltu $7,$3,$1 ;  if ($3 < $1) $7=1; else $7=0
```

- What are values of registers $4 - $7? Why?

$4 = 0 ; $5 = 1 ; $6 = 0 ; $7 = 0 ;

MIPS assembler register convention

Name	Number	Usage	Preserved across a call?
$zero	0	the value 0	n/a
$v0-$v1	2-3	return values	no
$a0-$a3	4-7	arguments	no
$t0-$t7	8-15	temporaries	no
$s0-$s7	16-23	saved	yes
$t18-$t19	24-25	temporaries	no
$sp	29	stack pointer	yes
$ra	31	return address	yes

- "caller saved" "callee saved"
- On Green Card in Column #2 at bottom

In class exercise: $s3=i, $s4=j, $s5=@A

```
Loop: addiu $s4,$s4,1     # j = j + 1
      sll   $t1,$s3,2     # $t1 = 4 * i
      addu  $t1,$t1,$s5   # $t1 = @ A[i]
      lw    $t0,0($t1)    # $t0 = A[i]
      slti  $t1,$t0,10    # $t1 = $t0 < 10
      beq   $t1,$0, Loop  # goto Loop
      addiu $s3,$s3,1     # i = i + 1
      slti  $t1,$t0, 0    # $t1 = $t0 <  0
      bne   $t1,$0, Loop  # goto Loop
```

```
do
      j = j + 1
while (______);
```

What C code properly fills in the blank in loop on right?

1: `A[i++] >= 10`
2: `A[i++] >= 10 |  A[i]   < 0`
3: `A[i]   >= 10 || A[i++] < 0`
4: `A[i++] >= 10 || A[i]   < 0`
5: `A[i]   >= 10 && A[i++] < 0`
6: None of the above

In class exercise: $s3=i, $s4=j, $s5=@A

```
Loop: addiu $s4,$s4,1     # j = j + 1
      sll   $t1,$s3,2     # $t1 = 4 * i
      addu  $t1,$t1,$s5   # $t1 = @ A[i]
      lw    $t0,0($t1)    # $t0 = A[i]
      slti  $t1,$t0,10    # $t1 = $t0 < 10
      beq   $t1,$0, Loop  # goto Loop if $t1 == 0 ($t0 >= 10)
      addiu $s3,$s3,1     # i = i + 1
      slti  $t1,$t0, 0    # $t1 = $t0 <  0
      bne   $t1,$0, Loop  # goto Loop if $t1 != 0 ($t0 < 0)
```

```
do
      j = j + 1
while (______);
```

What C code properly fills in the blank in loop on right?

1: `A[i++] >= 10`
2: `A[i++] >= 10 |  A[i]   < 0`
3: `A[i]   >= 10 || A[i++] < 0`
4: `A[i++] >= 10 || A[i]   < 0`
5: `A[i]   >= 10 && A[i++] < 0`
6: None of the above

Instruction Formats

- J-format: used for `j` and `jal`
- I-format: used for instructions with immediates, `lw` and `sw` (since the offset counts as an immediate), and the branches (`beq` and `bne`),
  - (but not the shift instructions; later)
- R-format: used for all other instructions

- It will soon become clear why the instructions have been partitioned in this way.

R-Format Instructions (1/2)

- Define "fields" of the following number of bits each: 6 + 5 + 5 + 5 + 5 + 6 = 32

6	5	5	5	5	6

- For simplicity, each field has a name:

opcode	rs	rt	rd	shamt	funct

R-Format Instructions (2/2)

- More fields:
  - `rs` (Source Register): *generally* used to specify register containing first operand
  - `rt` (Target Register): *generally* used to specify register containing second operand (note that name is misleading)
  - `rd` (Destination Register): *generally* used to specify register which will receive result of computation

J-Format Instructions (1/2)

- Define "fields" of the following number of bits each:

6 bits	26 bits

- As usual, each field has a name:

opcode	target address

- Key Concepts
  - Keep `opcode` field identical to R-format and I-format for consistency.
  - Combine all other fields to make room for large target address.

J-Format Instructions (2/2)

- Summary:
  - New PC = { PC[31..28], target address, 00 }
- Understand where each part came from!
- Note: In Verilog,
  { , , } means concatenation
  { 4 bits , 26 bits , 2 bits } = 32 bit address
  - { 1010, 11111111111111111111111111, 00 } = 10101111111111111111111111111100

I-Format Instructions

- Define "fields" of the following number of bits each:

6 bits	5 bits	5 bits	16 bits

- Each field has a name:

opcode	rs	rt	immediate

- Key Concepts
  - Keep `opcode` field identical to R-format and J-format for consistency.
  - Can specify jumps and address displacement within (roughly) $\pm 2^{15}$ range.

R-Format Example

- 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$_{ten}$

On Green Card: Format in column 1, opcodes in column 3

Green Card: OPCODES, BASE CONVERSION, ASCII (3)

MIPS opcode (31:26)	(1) MIPS funct (5:0)	(2) MIPS funct (5:0)	Binary	Deci-mal	Hexa-deci-mal	ASCII
(1)	sll	add.*f*	00 0000	0	0	NUL
j	srl	mul.*f*	00 0010	2	2	STX
lui	sync	floor.w.*f*	00 1111	15	f	SI
lbu	and	cvt.w.*f*	10 0100	36	24	$

(1) opcode(31:26) == 0
(2) opcode(31:26) == 17 ten (11 hex);
if fmt(25:21)==16 ten (10 hex) f = s (single);
if fmt(25:21)==17 ten (11 hex) f = d (double)

Note: 3-in-1 - Opcodes, base conversion, ASCII!

Green Card

- green card /n./ [after the "IBM System/360 Reference Data" card] A summary of an assembly language, even if the color is not green. Less frequently used now because of the decrease in the use of assembly language. Some green cards are actually booklets! For example,

  "I'll go get my green card so I can check the addressing mode for that instruction."

Image from Dave's Green Card Collection: http://www.planetmvs.com/greencard/

www.jargon.net

In class exercise

Which instruction has same representation as 35_{ten}?

A. add $0, $0, $0

B. subu $s0,$s0,$s0

C. lw $0, 0($0)

D. addi $0, $0, 35

E. subu $0, $0, $0

F. Trick question! Instructions are not numbers

❖Use Green Card handout to answer

In class exercise

Which instruction has same representation as 35_{ten}?

A. add $0, $0, $0

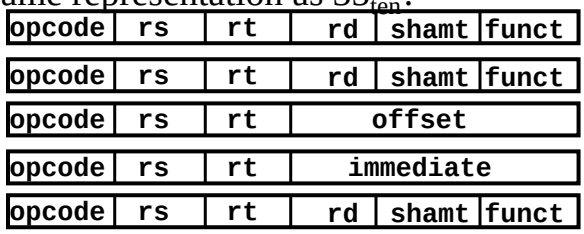
B. subu $s0,$s0,$s0

C. lw $0, 0($0)

D. addi $0, $0, 35

E. subu $0, $0, $0

F. Trick question! Instructions are not numbers



Registers numbers and names:
0: $0, 8: $t0, 9:$t1, ..15: $t7, 16: $s0, 17: $s1, .. 23: $s7

Opcodes and function fields (if necessary)

`add`: opcode = 0, funct = 32

`subu`: opcode = 0, funct = 35

`addi`: opcode = 8

`lw`: opcode = 35

In class exercise

Which instruction has same representation as 35_{ten}?

A. add $0, $0, $0

0	0	0	0	0	32

B. subu $s0,$s0,$s0

0	16	16	16	0	35

C. lw $0, 0($0)

35	0	0			0

D. addi $0, $0, 35

8	0	0			35

E. subu $0, $0, $0

0	0	0	0	0	35

F. Trick question! Instructions are not numbers

Registers numbers and names:
0: $0, 8: $t0, 9:$t1, ..15: $t7, 16: $s0, 17: $s1, .. 23: $s7

Opcodes and function fields (if necessary)

`add`: opcode = 0, funct = 32
`subu`: opcode = 0, funct = 35
`addi`: opcode = 8
`lw`: opcode = 35

Summary: Salient features of MIPS I

- **32-bit fixed format inst** (3 formats)
- **32 32-bit GPR** (R0 contains zero) and 32 FP registers (and HI LO)
  - partitioned by software convention
- **3-address, reg-reg arithmetic instr.**
- **Single address mode for load/store:** base+displacement
  - no indirection, scaled
- **16-bit immediate plus LUI**
- **Simple branch conditions**
  - compare against zero or two registers for =,≠
  - no integer condition codes
- **Delayed branch**
  - execute instruction after a branch (or jump) even if the branch is taken

  (Compiler can fill a delayed branch with useful work about 50% of the time)

Conclusion

- Instruction Set Architecture is the key abstraction between hardware designer and software developers
- Machine Organizations
  - Memory-to-Memory Machines
  - Accumulator
  - Stack
  - General Purpose Register Machines
    - MIPS ISA
      - General Purpose Register, Load/Store Machine
      - 32 registers, 32 bit operands, 32 bit main memory address space
      - 32 bit fixed length instructions - R, I and J instruction formats