Lecture 2: MIPS

SONGS ABOUT COMPUTER SCIENCE

The MIPS Instruction Set
Written by Walter Chang
To the tune of: The Major-General's Song
http://www.cs.utexas.edu/users/walter/cs-songbook/instruction_set.html
… …
There's sh and sb and lbu and blez and jal and then sltu
And of course there's and and add and srl and sub and things to do
With the MIPS instructions I am very nimble on my feet
And though I sing assembler but I am really not a geek

There's addu, ori, slti, swr, and bgez and jalr too
And loads of other fun instructions that were put in just for you
The MIPS instruction set is very simple to be memorized
Which will come in handy when you have your code to be optimized
… …

Topics

- MIPS **Assembly Language**
- RISC: Principles of good design
- R, I, J instruction formats
- Data access: Use registers; memory addressing
- Data process: Arithmetic instructions
- Programming constructs: Controlling flow of instructions
  - branches, if statement, loops, switch statement

Language of the machine, RISC, CISC

- Language of the machine
  - Instructions
  - More primitive than statements in higher level languages
  - Very restrictive formats
  - Design goals are:
- RISC: Reduced Instruction Set Computer
  - all instructions are simple, the same length
- also known as load / store architecture
- Another architecture: CISC (Complex …)
  - current example: Intel
- Is there a clear line distinguishing RISC and CISC?

Typical Operations (little change since 1960)

Data Movement	Load (from memory), Store (to memory) memory-to-memory move, register-to-register move input (from I/O device), output (to I/O device) push, pop (to/from stack)
Arithmetic	Add, Subtract, Multiply, Divide integer (binary + decimal) or FP
Shift	shift left/right (logical / arithmetic), rotate left/right
Logical	not, and, or, xor, set, clear
Control (J/Branch)	unconditional, conditional
Subroutine Linkage	call, return
Interrupt	trap, return
Synchronisation	test & set (atomic read-mod-write)
String	search, translate
Graphics	parallel subword ops (4 16bit add)

MIPS arithmetic

HP4 Section 2.2 P77-P80

- All instructions have 3 operands with fixed order: destination first. Simpler hardware!

Examples:

C assignment statement: `a = b + c`
Corresponding MIPS code: `add a,b,c`

C assignment statement: `a = b + c + d + e`
MIPS code:

```
add a,b,c #
add a,a,d #
add a,a,e #
```

1st principle of good design (more later, there are 4):
Simplicity favours regularity

MIPS arithmetic

- Simple statements

C code:

```
a = b + c + d;
e = f - a;
```

MIPS code:

```
add a,b,c #
add a,a,d #
sub e,f,a #
```

- A complex statement

C code: `f = (g + h) - (i + j);`

MIPS code:

```
add t0,g,h  # temp regs?
add t1,i,j  #
sub f,t0,t1 #
```


Registers as operands

- In MIPS arithmetic instructions operands must be registers
  - MIPS: 32 registers, each 32-bit wide, 32 bits is a word
- A complex statement again – PROPERLY coded:

C code: `f = (g + h) - (i + j);`

MIPS code:

```
add $t0,$s1,$s2 #
add $t1,$s3,$s4 #
sub $s0,$t0,$t1 #
```

- Compiler associates **variables** with registers
  - lots of variables – more registers?

2nd principle of good design:
Smaller is faster

Use immediate values – part 1/2

```
# program to calculate ? = (5 - 20) - (13 + 3)
# assumes: Numbers 5, -20, 13, 3 are in registers $s1 through $s4
        .data
        .globl mess
mess:   .asciiz "\nThe value of f is: " # string to print
                .text
        .globl main
main:                           # main has to be a global label
        addu $s7,$0,$ra         # save the return address in $s7
# the actual calculations follow:  # initialisation and move
                                   # immediate numbers to registers
        addi $s1,$0,5           # $s1 <= 5     <=>  s1=5; (C-like)
        addi $s2,$0,-20         # $s2 <= -20   <=>  s2=-20;
        addi $s3,$0,13          # $s3 <= 13    <=>  s3=13;
        addi $s4,$0,3           # $s4 <= 3     <=>  s4=3;
        add $t0,$s1,$s2         # 5 - 20       <=>  t0=s1+s2;
        add $t1,$s3,$s4         # 13 + 3       <=>  t1=s3+s4;
        sub $s0,$t0,$t1         # ? = (5 - 20) - (13 + 3)
                                #  <=>  s0=(s1+s2)-(s3+s4);
```


Use immediate values – part 2/2

```
        li $v0,4          # HP_AppA.pdf Page 44 or Appendix B in HP4
        la $a0,mess       # . . .
        syscall           # . . .
        li $v0,1          # . . .
        add $a0,$0,$s0    # . . .
        syscall
#Usual stuff at the end of the main
        addu $ra,$0,$s7  # restore the return address
        jr $ra           # return to the main program
```

Service	System call code	Arguments	Result
print_int	1	$a0 = integer	
print_float	2	$f12 = float	
print_double	3	$f12 = double	
print_string	4	$a0 = string	
read_int	5		integer (in $v0)
read_float	6		float (in $f0)
read_double	7		double (in $f0)
read_string	8	$a0 = buffer, $a1 = length	
sbrk	9	$a0 = amount	address (in $v0)
exit	10		
print_char	11	$a0 = char	

Use simple variables (see lab code) – part 1/2

```
# program to calculate f = (g + h) - (i + j)
# assumes: variables f through j are in registers $s0 through $s4
        .data
        .globl mess
mess:   .asciiz "\nThe value of f is: " # string to print
   f:   .word 0             # f = 0
   g:   .word 5             # simple/single variables
   h:   .word -20           # similar usage also as in lab 4 code
   i:   .word 13            # simplemem.s
   j:   .word 3
        .text
        .globl main
main:                       # main has to be a global label
        addu $s7,$0,$ra     # save the return address in $s7
# the actual calculations follow:
        lw $s1,g            # $s1 <= g = 5;
        lw $s2,h            # $s2 <= h = -20;
        lw $s3,i            # $s3 <= i = 13;
        lw $s4,j            # $s4 <= j = 3;
```

Caution: Avoid using *j* as variable in MIPS code as it may cause an error due to naming conflict with the jump instruction *j*.

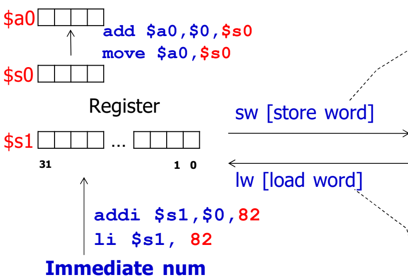
Use simple variables (see lab code) – part 1/2

```
        add $t0,$s1,$s2     # g + h
        add $t1,$s3,$s4     # ???
        sub $s0,$t0,$t1     # ???

        li $v0,4        # HP_AppA.pdf Page 44 or Appendix B in HP4
        la $a0,mess     # . . .
        syscall         # . . .
        li $v0,1        # . . .
        add $a0,$0,$s0  # . . .
        syscall
#Usual stuff at the end of the main
        addu $ra,$0,$s7 # restore the return address
        jr $ra          # return to the main program
```


MIPS data transfer

- Registers are adequate for **immediate numbers** and **simple variables**
- MIPS instructions to move data between registers and memory:



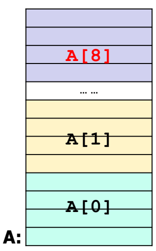
Memory access for storing:
sw $s0, f
source → destination
from register to memory

Memory access for loading:
lw $s1, g
destination ← source
from memory to register

Complex data structures - Array in memory

- Registers are adequate for **numbers** or **simple variables**
- **Arrays** may have more elements than registers available
- Example (**A**[…] in memory):

C code: `g = h + A[8];`



1. How to declare an array?
   int A[13];
   Name – Base address
   Size – Number of elements
   Type – Block size of single
2. How to locate and access an array element? **Index**
   A[8] Offset from base
3. Physical address **A[k]**
   Offset Base
   k*4 + A
4. How to define an array ***A*** ?
5. How to load ***A*** to register?
6. How to calculate ***k * 4*** ? …
7. Addressing syntax **x(y)**
   Offset(Base); B(O); 0(B+O)

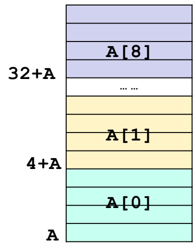
lw and sw Array element

HP4 Section 2.2 P83-P85

- Example (result in register, lw):

C code: `g = h + A[8];`
MIPS code:

```
lw $t0,32($s3)    #how to declare an array?
add $s1,$s2,$t0   #
```



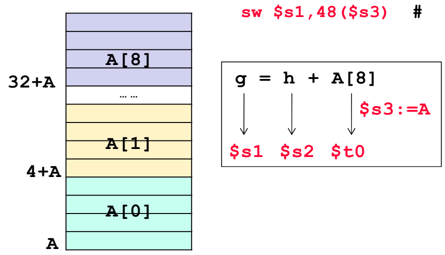
Address (leading cell)	Offset (byte)	Indices (word)
*k * 4 +A → k * 4 (A)*	*k * 4*	*k*th
4+A → 4(A)	1 * 4=4	k = 1
0+A → 0(A)	0 * 4=0	k = 0

lw and sw

- Example (result in register, lw):

C code: `A[12] = g;`
MIPS code:

```
lw $t0,32($s3)    #
add $s1,$s2,$t0   #
sw $s1,48($s3)    #
```



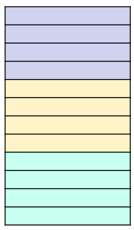
Using array index

- Example:

C code: `g = h + A[i];`

MIPS code:

```
# $t1 := i the word index; calculate offset 4*i
add $t1,$t1,$t1       # $t1 = i + i = 2i
add $t1,$t1,$t1       # $t1 = 2i + 2i = 4i
                      # adding replaces mult
# $s3 := A the base address; calculate 4*i + A
add $t1,$t1,$s3       # $t1 = address of A[i]
lw $t0,0($t1)         # $t0 gets A[i]
# $s2 := h
add $s1,$s2,$t0       # g (reg $s1) gets result
```



```
lw $t0, $t1($s3) # ?? $t1 + $s3
```


Spilling registers

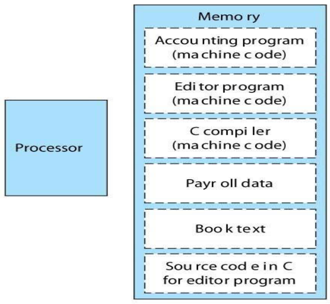
- Registers are faster than memory
  - smaller is faster
  - registers are faster to access and easier to use
- In RISC, data can only be operated on in registers ! ! !
- If more variables than registers: spilling registers
  - compiler must use registers efficiently for high performance

```
# $t1 := i the word index; calculate offset 4*i
add $t1,$t1,$t1       # $t1 = i + i = 2i
add $t1,$t1,$t1       # $t1 = 2i + 2i = 4i
# $s3 := A the base address; calculate 4*i + A
add $t1,$t1,$s3       # $t1 = address of A[i]
```

3rd principle of good design:
Good design demands good compromises

Stored Program Concept

- Programs are stored in memory
  - Instructions are represented as numbers (consisting of bits)
  - to be read or written just like data



Section 2.5, P101, 4th Ed

Translating machine language

See HP4, P134 and *instruction decoding.pdf* on vUWS

- Instructions, like registers and words are 32-bit long
- Each instruction consists of fields
  - each field is represented as a number, and has a specific meaning
- For example:
  - `add $t0,$s1,$s2`

0	17	18	8	0	32

  - the first and the last field in combination specify "add"
  - the second, third, and fourth field specify two source registers, and the destination register -- registers are represented as number between 0 and 31
  - the fifth field is unused in this instruction

MIPS Register Convention

- Important – keep a copy of this page!

Name	Register Number	Usage	Preserve on call?
$zero	0	constant 0 (hardware)	n.a.
$at	1	reserved for assembler	n.a.
$v0 - $v1	2-3	returned values	no
$a0 - $a3	4-7	arguments	yes
$t0 - $t7	8-15	temporaries	no
$s0 - $s7	16-23	saved values(declared variables)	yes
$t8 - $t9	24-25	temporaries	no
$k0, $k1	26, 27	reserved for OS kernel	n.a.
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return address (hardware)	yes

Instruction Formats: **R**, **I**, **J** types

- R-type Instruction format (R for Register)

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

- I-type Instruction format (I for Immediate)

op	rs	rt	constant
6 bits	5 bits	5 bits	16 bits

- J-type Instruction format (J for Jump)

op	address
6 bits	26 bits

EXERCISES

There will be many exercises.
(or: additional, NON GRADED homework)

The exercises WILL help you to *better*:
understand the material covered,
prepare you for labs,
prepare you for final exam.

Here is the first one:

Exercise example

- Can you figure out the code? (C followed by MIPS)

C code:

```
swap(int v[], int k)
{ int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

MIPS code:

```
swap:
    add $t0,$a1,$a1    #
    add $t0,$t0,$t0    # $t0 = 4k
    add $t0,$a0,$t0    # $t0 = address of v[k]
    lw $t1,0($t0)      # $t1 = v[k]
    lw $s0,4($t0)      # $s0 = v[k+1]
    sw $s0,0($t0)      # v[k] = $s0
    sw $t1,4($t0)      # v[k+1] = $t1
    jr $ra             # return
```


Controlling the flow of instructions

- Decision making instructions
  - alter the control flow (the "next" instruction)
  - distinguishes a computer from a simple calculator
- In a high level language - *if* statement, *go to* statement
- In an assembly language - *jumps*, conditional *branches*
- MIPS conditional branch instructions:

```
beq reg1,reg2,L1  # branch if equal
bne reg1,reg2,L1  # branch if not equal
```

C code:

```
    if (i==j) go to L1;
    f = g + h;
L1: f = f - i;
```

MIPS code :

```
    beq $s3,$s4,L1    #
    add $s0,$s1,$s2   #
                      #
L1: sub $s0,$s0,$s3   #
```

Avoid using ***j*** as variable in MIPS code as it may cause an error due to naming conflict with the jump instruction ***j***.

Control Flow

- We have *beq*, *bne*, what about Branch-if-less-than?

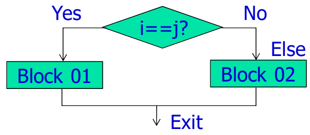
```
blt $s0,$s1,Less    # pseudoinstruction
```

- New instruction "set on less than":

```
slt $t0,$s0,$s1
bne $t0,$zero,Less  # reg 0 (=0)
```

If-then-else statement

- Example



Section 2.7, P107, 4th Ed

C code:

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

MIPS code:

```
      bne $s3,$s4,Else    # more efficient to test
                          # for opposite condition
      add $s0,$s1,$s2     # f = g + h
                          # skipped if i!=j
      j Exit              # jump (unconditional branch)
Else: sub $s0,$s1,$s2     # f = g - h
                          # skipped if i==j
Exit: …                   # some other instructions...
```

Caution: Avoid using ***j*** as variable in MIPS code as it may cause an error due to naming conflict with the jump instruction ***j***.

Loops {condition checking, looping block, occurrence updating}

- Simple loop:

C code (pseudo code):

```
Loop:  g = g + A[i];
       i = i + j;
       if ( i != h ) go to Loop
```

MIPS code:

```
Loop:  add $t1,$s3,$s3      # $t1 = 2i
       add $t1,$t1,$t1      # $t1 = 4i
       add $t1,$t1,$s5      # $t1 = address of A[i]
                            # $s5=array base address
       lw $t0,0($t1)        # $t0 = A[i]
       add $s1,$s1,$t0      # g = g + A[i]
       add $s3,$s3,$s4      # i = i + j
       bne $s3,$s2,Loop     # if i != h
       …                    # next instruction...
```


while loops

- while loops:

C code:

```
while (save[i] == k)
       i = i + j;
```

Caution: Avoid using ***j*** as variable in MIPS code as it may cause an error due to naming conflict with the jump instruction ***j***.

MIPS code:

```
Loop:  add $t1,$s3,$s3      # $t1 = 2i
       add $t1,$t1,$t1      # $t1 = 4i
       add $t1,$t1,$s5      # $t1 = address of save[i]
                            # $s5=array base address
       lw $t0,0($t1)        # $t0 <= save[i]
       bne $t0,$s2, Exit    # test condition, $s2 has k
       add $s3,$s3,$s4      # i = i + j
       j Loop               # keep looping
Exit:  …                    # next instruction...
```


Switch statement – home EXERCISE

```
switch (k) {
    case 0: f = i + j; break; /* k = 0 */
    case 1: f = g + h; break; /* k = 1 */
    case 2: f = g - h; break; /* k = 2 */
    case 3: f = i - j; break; /* k = 3 */
}
```

- Assume:
  - six variables f, g, h, i, j and k correspond to registers $s0 through to $s5;
  - register $t2 contains a value 4
- we may code the switch statement as a chain of if-then-else
- another solution: a jump address table
  - a table of addresses of a series of instruction sequences (an array of addresses)
  - Assume $t4 contains the address of the jump table
- we need an instruction to jump to an address contained in a register
  - "jump register" instruction: in MIPS: ***jr register***

Switch in assembly language

```
      slt $t3,$s5,$zero     # test if k<0
      bne $t3,$zero,Exit    # exit
      slt $t3,$s5,$t2       # test if k<4
      beq $t3,$zero,Exit    # exit if not
                # 0 < k < 4, ie. 0, 1, 2 and 3
      add $t1,$s5,$s5       # $t1 = 2k
      add $t1,$t1,$t1       # $t1 = 4k
      add $t1,$t1,$t4       # $t1 = offset to jump table
      lw $t0,0($t1)         # $t0 = jump-table[k]
      jr $t0  # jump to appropriate case in the switch statement
L0:   add $s0,$s3,$s4       # k=0, so f = i + j
      j Exit                # break
L1:   add $s0,$s1,$s2       # k=1, so f = g + h
      j Exit                # break
L2:   sub $s0,$s1,$s2       # k=0, so f = g - h
      j Exit                # break
L3:   sub $s0,$s3,$s4       # k=0, so f = i - j
Exit: ...                   # some instruction
```


Revision

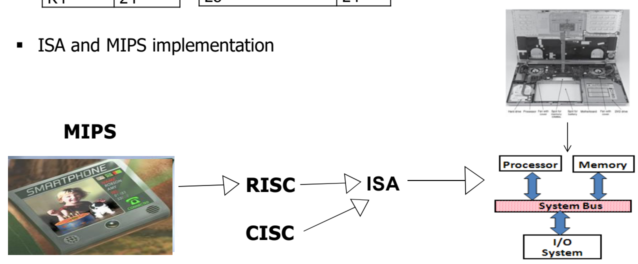
- Given the register and memory values in the tables below (with dummy data for easy calculation), work out the contents of registers in the instructions.

Register	Value
R1	12
R2	16
R3	20
R4	24

Memory Location	Value
16	20
20	12
24	16
28	24

lw R3, 12(R1)
addi R2, R3, 12

- ISA and MIPS implementation



ASCII TABLE

See *ascii_chart.pdf* on vUWS

DEC	HEX	CHAR		DEC	HEX	CHAR	DEC	HEX	CHAR	DEC	HEX	CHAR
0	00	^@	NUL	32	20	SPC	64	40	@	96	60	`
1	01	^A	SOH	33	21	!	65	41	A	97	61	a
2	02	^B	STX	34	22	"	66	42	B	98	62	b
3	03	^C	ETX	35	23	#	67	43	C	99	63	c
4	04	^D	EOT	36	24	$	68	44	D	100	64	d
5	05	^E	ENQ	37	25	%	69	45	E	101	65	e
6	06	^F	ACK	38	26	&	70	46	F	102	66	f
7	07	^G	BEL	39	27	'	71	47	G	103	67	g
8	08	^H	BS	40	28	(	72	48	H	104	68	h
9	09	^I	HT	41	29	)	73	49	I	105	69	i
10	0A	^J	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	^K	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	^L	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	^M	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	^N	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	^O	SI	47	2F	/	79	4F	O	111	6F	o
16	10	^P	DLE	48	30	0	80	50	P	112	70	p
17	11	^Q	DC1	49	31	1	81	51	Q	113	71	q
18	12	^R	DC2	50	32	2	82	52	R	114	72	r
19	13	^S	DC3	51	33	3	83	53	S	115	73	s
20	14	^T	DC4	52	34	4	84	54	T	116	74	t
21	15	^U	NAK	53	35	5	85	55	U	117	75	u
22	16	^V	SYN	54	36	6	86	56	V	118	76	v
23	17	^W	ETB	55	37	7	87	57	W	119	77	w
24	18	^X	CAN	56	38	8	88	58	X	120	78	x
25	19	^Y	EM	57	39	9	89	59	Y	121	79	y
26	1A	^Z	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	^[	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	^\	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	^]	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	^^	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	^_	US	63	3F	?	95	5F	_	127	7F	DEL

Recommended readings

General Data	UnitOutline | LearningGuide | Teaching Schedule | Aligning Assessments |
Extra Materials	ascii_chart.pdf | bias_representation.pdf | HP_AppA.pdf | instruction decoding.pdf | masking help.pdf | PCSpim.pdf | PCSpim Portable Version | Library materials

PH6, §2.2-§2.3, P69: Operations and Operands
PH5, §2.2-§2.3, P63: Operations and Operands
PH4, §2.2-§2.3, P78: Operations and Operands

PH6, §2.2-§2.3, §2.5: 1st-3rd Principle of hardware design
PH5, §2.2-§2.3, §2.5: 1st-3rd Principle of hardware design
P65: Design Principle 1
P67: Design Principle 2
P83: Design Principle 3
PH4, §2.2-§2.3, §2.5, P79-P97: 1st-4th Principle of hardware design
P79: Design Principle 1
P81: Design Principle 2
P86: Design Principle 3
P97: Design Principle 4

PH6, §2.5, P86: pay attention to Stored-Program Concept
PH5, §2.5, P86: pay attention to Stored-Program Concept
PH4, §2.5, P101: pay attention to Stored-Program Concept

PH6, §2.7, P96: Understand basic control structures
PH5, §2.7, P90-P96: Understand basic control structures
PH4, §2.7, P105-P111: Understand basic control structures

HP_AppA.pdf -> A-21: Memory layout

HP_AppA.pdf-> A-44: System services

Text readings are listed in Teaching Schedule and Learning Guide

PH6 (PH5 & PH4 also suitable): check whether eBook available on library site

PH6: companion materials (e.g. online sections for further readings)
https://www.elsevier.com/books-and-journals/book-companion/9780128201091

PH5: companion materials (e.g. online sections for further readings)
http://booksite.elsevier.com/9780124077263/?ISBN=9780124077263