Computer Architecture and Organization

Chapter 3 Instructions: Language of the Machine

Henk Corporaal / Ben Juurlink
Delft University of Technology
Spring 2000
Email: h.corporaal@et.tudelft.nl
URL : http://cardit.et.tudelft.nl/~heco/courses/et2-010

Topics

- Instructions & MIPS instruction set
- Where are the operands ?
- Machine language
- Assembler
- Translating C statements into Assembler
- Other architectures:
  - PowerPC
  - Intel 80x86
- More complex stuff, like:
  - while statement
  - switch statement
  - procedure / function (leaf and nested)
  - stack
  - linking object files

Instructions:

- Language of the Machine
- More primitive than higher level languages
  e.g., no sophisticated control flow
- Very restrictive
  e.g., MIPS Arithmetic Instructions
- We'll be working with the MIPS instruction set architecture
  - similar to other architectures developed since the 1980's
  - used by NEC, Nintendo, Silicon Graphics, Sony

Design goals: maximize performance and minimize cost, reduce design time, reduce energy consumption

Main Types of Instructions

- Arithmetic
  - Integer
  - Floating Point
- Memory access instructions
  - Load & Store
- Control flow
  - Jump
  - Conditional Branch
  - Call & Return

MIPS arithmetic

- Most instructions have 3 operands
- Operand order is fixed (destination first)

Example:

C code: `A = B + C`

MIPS code: `add $s0, $s1, $s2`

($s0, $s1 and $s2 are associated with variables by compiler)

MIPS arithmetic

- Design Principle: *simplicity favors regularity*. Why?
- Of course this complicates some things...

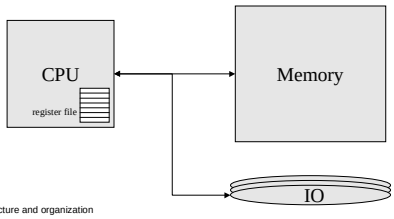
C code:
```
A = B + C + D;
E = F - A;
```

MIPS code:
```
add $t0, $s1, $s2
add $s0, $t0, $s3
sub $s4, $s5, $s0
```

- Operands must be registers, only 32 registers provided
- Design Principle: *smaller is faster*. Why?

Registers vs. Memory

- Arithmetic instructions operands must be registers,
  — only 32 registers provided
- Compiler associates variables with registers
- What about programs with lots of variables ?

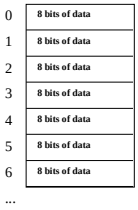


Register allocation

- Compiler tries to keep as many variables in registers as possible
- Some variables can not be allocated
  - large arrays (too few registers)
  - aliased variables (variables accessible through pointers in C)
  - dynamic allocated variables
    - heap
    - stack
- Compiler may run out of registers => ***spilling***

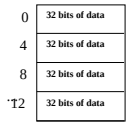
Memory Organization

- Viewed as a large, single-dimension array, with an address.
- A memory address is an index into the array
- "Byte addressing" means that the index points to a byte of memory.



Memory Organization

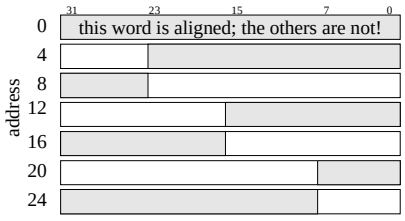
- Bytes are nice, but most data items use larger "words"
- For MIPS, a word is 32 bits or 4 bytes.



Registers hold 32 bits of data

- 2^{32} bytes with byte addresses from 0 to 2^{32}-1
- 2^{30} words with byte addresses 0, 4, 8, ... 2^{32}-4

Memory layout: Alignment



- Words are aligned
  i.e., what are the least 2 significant bits of a word address?

Instructions

- Load and store instructions
- Example:

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

MIPS code:
```
lw $t0, 32($s3)
add $t0, $s2, $t0
sw $t0, 32($s3)
```

- Store word has destination last
- Remember arithmetic operands are registers, not memory!

Our First C Example

- Can we figure out the code?

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

➡

```
swap:
   muli $2 , $5, 4
   add  $2 , $4, $2
   lw   $15, 0($2)
   lw   $16, 4($2)
   sw   $16, 0($2)
   sw   $15, 4($2)
   jr   $31
```

Explanation:
index k : $5
base address of v: $4
address of v[k] is $4 + 4.$5

So far we've learned:

- MIPS
  - — loading words but addressing bytes
  - — arithmetic on registers only
- Instruction / Meaning

```
add $s1, $s2, $s3    $s1 = $s2 + $s3
sub $s1, $s2, $s3    $s1 = $s2 – $s3
lw $s1, 100($s2)     $s1 = Memory[$s2+100]
sw $s1, 100($s2)     Memory[$s2+100] = $s1
```


Machine Language

- Instructions, like registers and words of data, are also 32 bits long
  - Example: `add $t0, $s1, $s2`
  - Registers have numbers: `$t0=9, $s1=17, $s2=18`
- Instruction Format:

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
000000	10001	10010	01000	00000	100000
op	rs	rt	rd	shamt	funct

- *Can you guess what the field names stand for?*

Machine Language

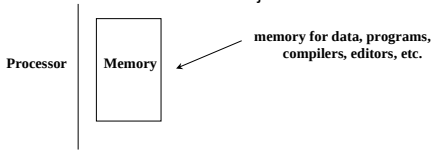
- Consider the load-word and store-word instructions,
  - What would the regularity principle have us do?
  - New principle: *Good design demands a compromise*
- Introduce a new type of instruction format
  - I-type for data transfer instructions
  - other format was R-type for register
- Example: `lw $t0, 32($s2)`

35	18	9	32
op	rs	rt	16 bit number

- *Where's the compromise?*
- Study example page 119-120

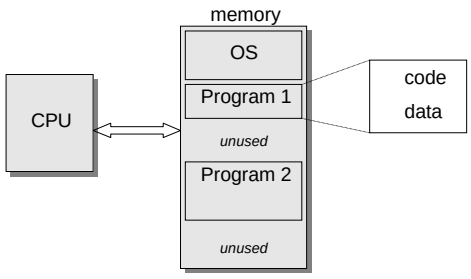
Stored Program Concept

- Instructions are bits
- Programs are stored in memory
  - — to be read or written just like data



Fetch & Execute Cycle
- Instructions are fetched and put into a special register
- Bits in the register "control" the subsequent actions
- Fetch the "next" instruction and continue

Stored Program Concept



Control

- Decision making instructions
  - alter the control flow,
  - i.e., change the "next" instruction to be executed
- MIPS conditional branch instructions:

```
bne $t0, $t1, Label
beq $t0, $t1, Label
```

- Example: if (i==j) h = i + j;

```
        bne $s0, $s1, Label
        add $s3, $s0, $s1
Label:  ....
```


Control

- MIPS unconditional branch instructions:
  `j label`
- Example:

```
if (i!=j)           beq $s4, $s5, Lab1
    h=i+j;          add $s3, $s4, $s5
else                j Lab2
    h=i-j;      Lab1:sub $s3, $s4, $s5
                Lab2:...
```

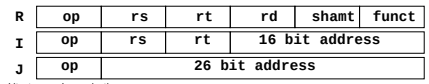
- *Can you build a simple for loop?*

So far:

- Instruction Meaning

```
add $s1,$s2,$s3  $s1 = $s2 + $s3
sub $s1,$s2,$s3  $s1 = $s2 – $s3
lw $s1,100($s2)  $s1 = Memory[$s2+100]
sw $s1,100($s2)  Memory[$s2+100] = $s1
bne $s4,$s5,L    Next instr. is at Label if $s4 ° $s5
beq $s4,$s5,L    Next instr. is at Label if $s4 = $s5
j Label          Next instr. is at Label
```

- Formats:



Control Flow

- We have: beq, bne, what about Branch-if-less-than?
- New instruction:

```
slt $t0, $s1, $s2
```

```
if  $s1 < $s2 then
        $t0 = 1
else
        $t0 = 0
```

- Can use this instruction to build "`blt $s1, $s2, Label`"
  — can now build general control structures
- Note that the assembler needs a register to do this,
  — there are policy of use conventions for registers

Used MIPS Conventions

Name	Register number	Usage
$zero	0	the constant value 0
$v0-$v1	2-3	values for results and expression evaluation
$a0-$a3	4-7	arguments
$t0-$t7	8-15	temporaries
$s0-$s7	16-23	saved
$t8-$t9	24-25	more temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address

Constants

- Small constants are used quite frequently (50% of operands) e.g., A = A + 5;
  B = B + 1;
  C = C - 18;
- Solutions? Why not?
  - put 'typical constants' in memory and load them.
  - create hard-wired registers (like $zero) for constants like one.
- MIPS Instructions:

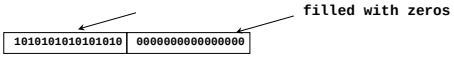
```
addi $29, $29, 4
slti $8,  $18, 10
andi $29, $29, 6
ori  $29, $29, 4
```

- How do we make this work?

How about larger constants?

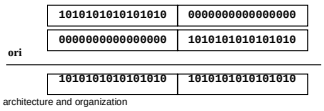
- We'd like to be able to load a 32 bit constant into a register
- Must use two instructions, new "load upper immediate" instruction

```
lui $t0, 1010101010101010
```



- Then must get the lower order bits right, i.e.,

```
ori $t0, $t0, 1010101010101010
```



Assembly Language vs. Machine Language

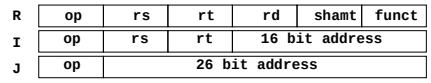
- Assembly provides convenient symbolic representation
  - much easier than writing down numbers
  - e.g., destination first
- Machine language is the underlying reality
  - e.g., destination is no longer first
- Assembly can provide '*pseudoinstructions*'
  - e.g., "move $t0, $t1" exists only in Assembly
  - would be implemented using "add $t0,$t1,$zero"
- When considering performance you should count real instructions

Other Issues

- Things not yet covered:
  - support for procedures
  - linkers, loaders, memory layout
  - stacks, frames, recursion
  - manipulating strings and pointers
  - interrupts and exceptions
  - system calls and conventions
- Some of these we'll talk about later
- We've focused on architectural issues
  - basics of MIPS assembly language and machine code
  - we'll build a processor to execute these instructions.

Overview of MIPS

- simple instructions all 32 bits wide
- very structured, no unnecessary baggage
- only three instruction formats



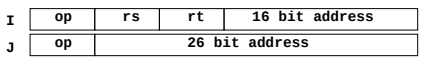
- rely on compiler to achieve performance
  — what are the compiler's goals?
- help compiler where we can

Addresses in Branches and Jumps

- Instructions:

```
bne $t4,$t5,Label   Next instruction is at Label if $t4 ° $t5
beq $t4,$t5,Label   Next instruction is at Label if $t4 = $t5
j Label             Next instruction is at Label
```

- Formats:



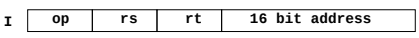
- Addresses are not 32 bits
  — How do we handle this with load and store instructions?

Addresses in Branches

- Instructions:

```
bne $t4,$t5,Label   Next instruction is at Label if $t4°$t5
beq $t4,$t5,Label   Next instruction is at Label if $t4=$t5
```

- Formats:



- Could specify a register (like lw and sw) and add it to address
  - use Instruction Address Register (PC = program counter)
  - most branches are local (principle of locality)
- Jump instructions just use high order bits of PC
  - address boundaries of 256 MB

To summarize:

MIPS operands

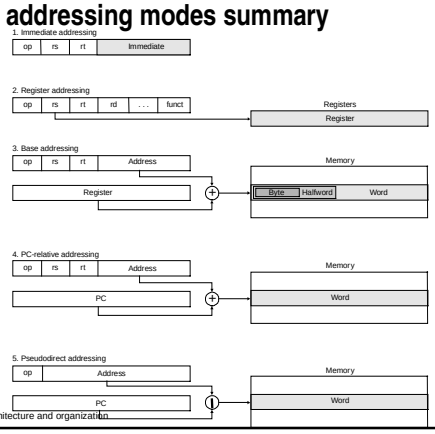
Name	Example	Comments
	$s0-$s7, $t0-$t9, $zero,	Fast locations for data. In MIPS, data must be in registers to perform
32 registers	$a0-$a3, $v0-$v1, $gp,	arithmetic. MIPS register $zero always equals 0. Register $at is
	$fp, $sp, $ra, $at	reserved for the assembler to handle large constants.
	Memory[0],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

To summarize:

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1, $s2, $s3`	$s1 = $s2 + $s3	Three operands; data in registers
	subtract	`sub $s1, $s2, $s3`	$s1 = $s2 - $s3	Three operands; data in registers
	add immediate	`addi $s1, $s2, 100`	$s1 = $s2 + 100	Used to add constants
Data transfer	load word	`lw $s1, 100($s2)`	$s1 = Memory[$s2 + 100]	Word from memory to register
	store word	`sw $s1, 100($s2)`	Memory[$s2 + 100] = $s1	Word from register to memory
	load byte	`lb $s1, 100($s2)`	$s1 = Memory[$s2 + 100]	Byte from memory to register
	store byte	`sb $s1, 100($s2)`	Memory[$s2 + 100] = $s1	Byte from register to memory
	load upper immediate	`lui $s1, 100`	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
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 less than immediate	`slti $s1, $s2, 100`	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare less than constant
Uncondi-tional 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

MIPS addressing modes summary



Alternative Architectures

- Design alternative:
  - provide more powerful operations
  - goal is to reduce number of instructions executed
  - danger is a slower cycle time and/or a higher CPI
- Sometimes referred to as “RISC vs. CISC”
  - virtually all new instruction sets since 1982 have been RISC
  - VAX: minimize code size, make assembly language easy *instructions from 1 to 54 bytes long!*
- We’ll look at PowerPC and 80x86

PowerPC

- Indexed addressing
  - example: `lw $t1,$a0+$s3` `#$t1=Memory[$a0+$s3]`
  - What do we have to do in MIPS?
- Update addressing
  - update a register as part of load (for marching through arrays)
  - example: `lwu $t0,4($s3)` `#$t0=Memory[$s3+4];$s3=$s3+4`
  - What do we have to do in MIPS?
- Others:
  - load multiple/store multiple
  - a special counter register “`bc Loop`” *decrement counter, if not 0 goto loop*

80x86 history

- 1978: The Intel 8086 is announced (16 bit architecture)
- 1980: The 8087 floating point coprocessor is added
- 1982: The 80286 increases address space to 24 bits, +instructions
- 1985: The 80386 extends to 32 bits, new addressing modes
- 1989-1995: The 80486, Pentium, Pentium Pro add a few instructions (mostly designed for higher performance)
- 1997: MMX is added

“This history illustrates the impact of the “golden handcuffs” of compatibility

“adding new features as someone might add clothing to a packed bag”

“an architecture that is difficult to explain and impossible to love”

A dominant architecture: 80x86

- See your textbook for a more detailed description
- Complexity:
  - Instructions from 1 to 17 bytes long
  - one operand must act as both a source and destination
  - one operand can come from memory
  - complex addressing modes
    e.g., "base or scaled index with 8 or 32 bit displacement"
- Saving grace:
  - the most frequently used instructions are not too difficult to build
  - compilers avoid the portions of the architecture that are slow

"what the 80x86 lacks in style is made up in quantity, making it beautiful from the right perspective"

Summary (so far)

- Instruction complexity is only one variable
  - lower instruction count vs. higher CPI / lower clock rate
- Design Principles:
  - simplicity favors regularity
  - smaller is faster
  - good design demands compromise
  - make the common case fast
- Instruction set architecture
  - a very important abstraction indeed!

More complex stuff

- While statement
- Case/Switch statement
- Procedure
  - leaf
  - non-leaf / recursive
- Stack
- Memory layout
- Characters, Strings
- Arrays versus Pointers
- Starting a program
  - Linking object files

While statement

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

```
Loop:  muli $t1,$s3,4      # calculate address of
       add  $t1,$t1,$s6    # save[i]
       lw   $t0,0($s1)
       bne  $t0,$s5,Exit
       add  $s3,$s3,$s4
       j    Loop
Exit:
```


Case/Switch statement

C Code:

```
switch (k) {
   case 0: f=i+j; break;
   case 1: ............;
   case 2: ............;
   case 3: ............;
}
```

Assembler Code:

1. test if k inside 0-3
2. calculate address of jump table location
3. fetch jump address and jump
4. code for all different cases (with labels L0-L3)

Data: jump table

address L0
address L1
address L2
address L3

Compiling a leaf Procedure

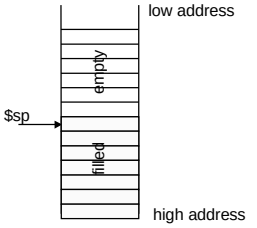
C code

```
int leaf_example (int g, int h, int i, int j)
{
  int f;
  f = (g+h)-(i+j);
  return f;
}
```

Assembler code

```
leaf_example: save registers changed by callee
              code for expression 'f = ....'
                 (g is in $a0, h in $a1, etc.)
              put return value in $v0
              restore saved registers
              jr $ra
```

Using a Stack



Save $s0 and $s1:

```
subi $sp,$sp,8
lw   $s0,4($sp)
lw   $s1,0($sp)
```

Restore $s0 and $s1:

```
sw   $s0,4($sp)
sw   $s1,0($sp)
addi $sp,$sp,8
```

Convention: $ti registers do not have to be saved and restored by callee

They are scratch registers

Compiling a non-leaf procedure

C code of 'recursive' factorial

```
int fact (int n)
{
  if (n<1) return (1)
     else return (n*fact(n-1));
}
```

Factorial: n! = n* (n-1)!

0! = 1

Compiling a non-leaf procedure

For non-leaf procedure

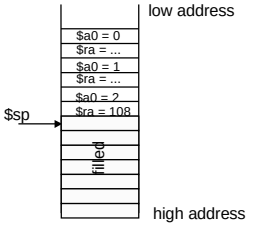
- save arguments registers (if used)
- save return address ($ra)
- save callee used registers
- create stack space for local arrays and structures (if any)

Compiling a non-leaf procedure

Assembler code for 'fact'

```
fact: subi $sp,$sp,8
      sw   $ra,4($sp)
      sw   $a0,0($sp)
      slti $to,$a0,1     # test for n<1
      beq  $t0,$zero,L1  # if n>= 1 goto L1
      addi $v0,$zero,1   # return 1
      addi $sp,$sp,8     # check this !
      jr   $ra
L1:   subi $a0,$a0,1
      jal  fact          # call fact with (n-1)
      lw   $a0,0($sp)
      lw   $ra,4($sp)
      addi $sp,$sp,8
      mul  $v0,$a0,$v0   # return n*fact(n-1)
      jr   $ra
```


How does the stack look?




```
100 addi $a0,$zero,2
104 jal fact
108 ....
```

Note: no callee regs are used

Beyond numbers: characters

- Characters are often represented using the ASCII standard
- ASCII = American Standard COde for Information Interchange
- See table 3.15, page 142
- Note: value(a) - value(A) = 32
  value(z) - value(Z) = 32

Beyond numbers: Strings

- A string is a sequence of characters
- Representation alternatives for “aap”:
  - including length field: 3’a’’a’’p’
  - separate length field
  - delimiter at the end: ‘a’’a’’p’0 (Choice of language C !!)

Discuss C procedure ‘strcpy’

```
void strcpy (char x[], char y[])
{
  int i;
  i=0;
  while ((x[i]=y[i]) != 0)   /* copy and test byte */
        i=i+1;
}
```


String copy: strcpy

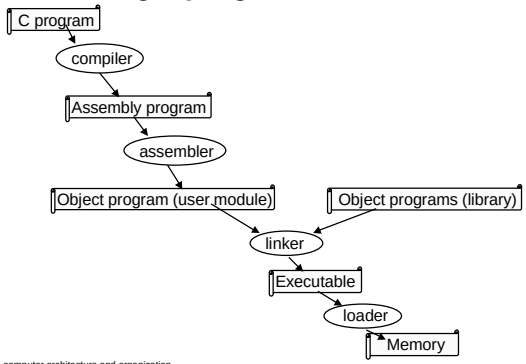
```
strcpy: subi $sp,$sp,4
        sw   $s0,0($sp)
        add  $s0,$zero,$zero  # i=0
L1:     add  $t1,$a1,$s0      # address of y[i]
        lb   $t2,0($t1)       # load y[i] in $t2
        add  $t3,$a0,$s0      # similar address for x[i]
        sb   $t2,0($t3)       # put y[i] into x[i]
        addi $s0,$s0,1
        bne  $t2,$zero,L1     # if y[i]!=0 go to L1
        lw   $s0,0($sp)       # restore old $s0
        add1 $sp,$sp,4
        jr   $ra
```

Note: strcpy is a leaf-procedure; no saving of args and return address required

Starting a program

- Compile C program
- Assemble
- Link
  - insert library code
  - determine addresses of data and instruction labels
  - relocation: patch addresses
- Load into memory
  - load text (code)
  - load data (global data)
  - initialize $sp, $gp
  - copy parameters to the main program onto the stack
  - jump to ‘start-up’ routine
    - copies parameters into $ai registers
    - call main

Starting a program



Arrays versus pointers

Array version:

```
clear1 (int array[], int size)
{
  int i;
  for (i=0; i<size; i=i+1)
    array[i]=0;
}
```

Two programs which initialize an array to zero

Pointer version:

```
clear2 (int *array, int size)
{
  int *p;
  for (p=&array[0]; p<&array[size]; p=p+1)
    *p=0;
}
```


Arrays versus pointers

- Compare the assembly result on page 174
- Note the size of the loop body:
  - Array version: 7 instructions
  - Pointer version: 4 instructions
- Pointer version much faster !
- Clever compilers perform pointer conversion themselves

Exercises

- Make from chapter three the following exercises:
  - 3.1 - 3.6
  - 3.8
  - 3.16 (calculate CPI for gcc only)
  - 3.19, 3.20