MIPS Architecture

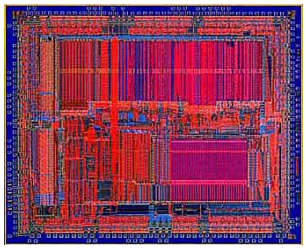
Many sheets copied from Andreas Klappenecker.
Additional information "MIPS Assembly Language programming", Rober L. Britton (Prentice Hall, ISBN 0-13-142044-5) and many sources on internet

MIPS Design Paradigms

- Simplicity favors regularity
  - all instructions single size
  - three register operands in arithmetic instr.
  - keep register fields in the same place
- Smaller is faster
  - 32 registers
- Make the common case fast
  - PC-relative addressing for conditional branches

MIPS R2000

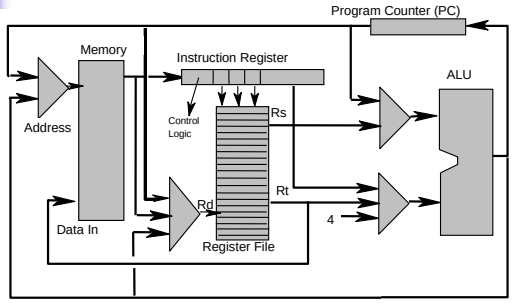
- Several firsts:
  - First RISC microprocessor
  - First microprocessor to provide integrated support for instruction & data cache
  - First pipelined microprocessor
- Implemented in 1985
  - 125,000 transistors
  - 5-8 MIPS



Basic Functional Components

- Control unit
- Register file
- Arithmetic logic unit (ALU)
- Program counter (PC)
- Memory
- Instruction register (IR)

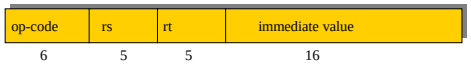
Simplified Datapath Diagram



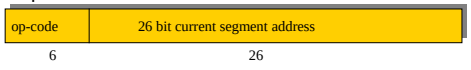
Instruction Word Formats

- Register format

- Immediate format

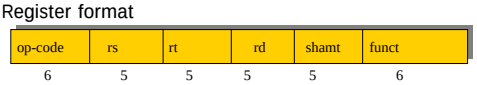


- Jump format



Register Format (R-Format)

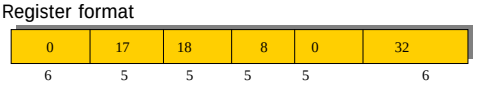
- Register format



- op: *basic operation of instruction*
- funct: *variant of instruction*
- rs: *first register source operand*
- rt: *second register source operand*
- rd: *register destination operand*
- shamt: *shift amount*

R-Format Example

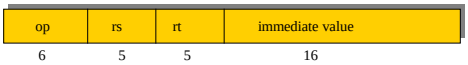
- Register format



- (op, funct)=(0,32): add
- rs=17: first source operand is $s1
  - Note: de 32 register addresses have names starting with '$'
- rt=18: second source operand is $s2
- Rd=8: register destination is $t0
- add $t0, $s1, $s2

Immediate Format (I-Format)

Immediate format

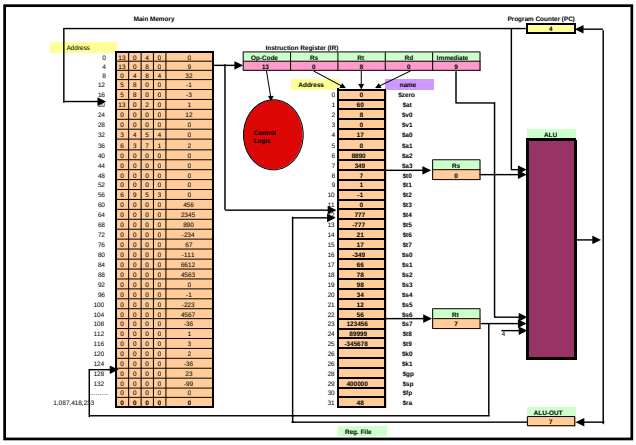


- op determines the instruction
- rs is the source register
- rt is the destination register
- 16bit immediate value

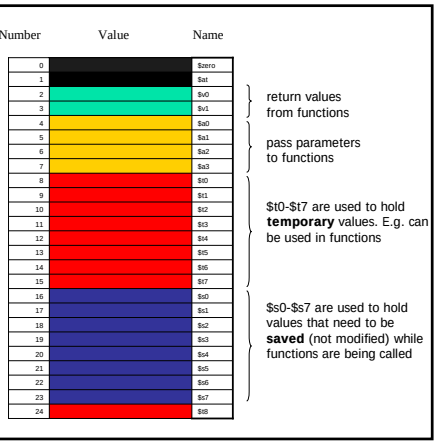
I-Format Example

Immediate format

- op=8 means addi
- rs=29 means source register is $sp
- rt=29 means $sp is destination register
- immediate value = 4
- addi $sp, $sp, 4



Registers



Some problems

- Immediate field is 16 bits. How can we load a register with 32 bits?
- The result of multiplication of two 32 bits is 64 bits. Where is the result?

How can we load a 32bit constant?

- How can we load $s0 with
  - 0000 0000 0011 1101 0000 1001 0000 0000
- Load upper 16bits with
  - lui $s0, Ox003d so that $s0 contains (16 bits on the right)
  - 0000 0000 0011 1101 0000 0000 0000 0000
- Add immediate 16bit value to complete load
  - addi $s0, $s0, 0x0900 so that $s0 contains value
  - 0000 0000 0011 1101 0000 1001 0000 0000

Where are my multiply result?

- Multiple 32 bits x 32 bits ➔ 64 bits
  - mult $a1, $s1
  - mfhi $v0
  - mflo $v1

MIPS Addressing Modes

- Register addressing
- Base displacement addressing
- Immediate addressing
- PC-relative addressing
  - address is the sum of the PC and a constant in the instruction
- Pseudo-direct addressing
  - jump address is 26bits of instruction concatenated with upper bits of PC

MIPS Assembly Language

MIPS Assembly Instructions

- add $t0, $t1, $t2 # $t0=$t1+$t2
- sub $t0, $t1, $t2 # $t0=$t1-$t2

- lw $t1, a_addr # $t1=Mem[a_addr]
- sw $s1, a_addr # Mem[a_addr]=$t1

Assembler directives

- .text assembly instructions follow
- .data data follows
- .globl globally visible label
  = symbolic address

Hello World!

```
        .text                   # code section
        .globl main
main:   li $v0, 4               # system call for print string
        la $a0, str             # load address of string to print
        syscall                 # print the string
        li $v0, 10              # system call for exit
        syscall                 # exit
        .data
str:    .asciiz "Hello world!\n"  # NUL terminated string, as in C
```

Addressing modes

```
lw $s1, addr        # load $s1 from addr
lw $s1, 8($s0)      # $s1 = Mem[$s0+8]
```

register $s0 contains the base address

access the address ($s0)

possibly add an offset 8($s0)

Load and move instructions

```
la  $a0, addr       # load address addr into $a0 (macro)
li   $a0, 12        # load immediate $a0 = 12
lbu $a0, c($s1)     # load $a0 = zero_extended(Mem[$s1+c])
lb $a0, c($s1)      # load $a0 = sign_extended(Mem[$s1+c])
lw $a0, c($s1)      # load word
move $s0, $s1       # $s0 = $s1
```

Control Structures

Assembly language has very few control structures:

- Branch instructions if cond then goto label
- Jump instructions goto label

We can build while loops, for loops, repeat-until loops, if-then-else structures from these primitives

Branch instructions

```
beqz $s0, label          if $s0==0     goto label
bnez $s0, label          if $s0!=0     goto label
bge  $s0, $s1, label     if $s0>=$s1   goto label
ble  $s0, $s1, label     if $s0<=$s1   goto label
blt  $s0, $s1, label     if $s0<$s1    goto label
beq  $s0, $s1, label     if $s0==$s1   goto label
bgez  $s0, $s1, label    if $s0>=0     goto label
```

if-then-else structures

```
if ($t0==$t1) then /* blockA */ else /* blockB */

        beq $t0, $t1, blockA
        j blockB
blockA: ... instructions of then block ...
        j exit
blockB: ... instructions of else block ...
exit:   ... subsequent instructions ...
```

repeat-until loop

```
repeat ... until $t0>$t1

loop: ... instructions of loop ...
        ble $t0, $t1, loop        # if $t0<=$t1 goto loop
```

Other loop structures are similar...

System calls

- load argument registers
- load call code
- syscall

```
li  $a0, 10     # load argument $a0=10
li  $v0, 1      # call code to print integer
syscall         # print $a0
```

SPIM system calls

procedure	code $v0	argument
print int	1	$a0 contains number
print float	2	$f12 contains number
print double	3	$f12 contains number
print string	4	$a0 address of string

SPIM system calls

procedure	code $v0	result
read int	5	res returned in $v0
read float	6	res returned in $f0
read double	7	res returned in $f0
Exit	10	

Example programs

- Loop printing integers 1 to 10

```
1
2
3
```

- Increasing array elements by 5

```
for(i=0; i<len; i++) {
  a[i] = a[i] + 5;
}
```

Print numbers 1 to 10

```
main:   li $s0, 1               # $s0 = loop counter
        li $s1, 10              # $s1 = upper bound of loop
loop:   move $a0, $s0           # print loop counter $s0
        li $v0, 1
        syscall
        li $v0, 4               # print "\n"
        la $a0, linebrk         # linebrk: .asciiz "\n"
        syscall
        addi $s0, $s0, 1        # increase counter by 1
        ble $s0, $s1, loop      # if ($s0<=$s1) goto loop
        li $v0, 10              # exit
        syscall
        .data
linebrk: .asciiz "\n"
```

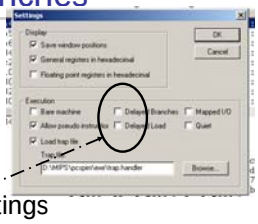
Increase array elements by 5

```
        .text
        .globl main
main:   la   $t0, Aaddr         # $t0 = pointer to array A
        lw   $t1, len           # $t1 = length (of array A)
        sll  $t1, $t1, 2        # $t1 = 4*length (sll=shift left logical)
        add  $t1, $t1, $t0      # $t1 = address(A)+4*length
loop:   lw   $t2, 0($t0)        # $t2 = A[i]
        addi $t2, $t2, 5        # $t2 = $t2 + 5
        sw   $t2, 0($t0)        # A[i] = $t2
        addi $t0, $t0, 4        # i = i+1
        bne  $t0, $t1, loop     # if $t0<$t1 goto loop
        .data
Aaddr:  .word 0,2,1,4,5         # array with 5 elements
len:    .word 5
```

Procedures

- jal addr
  - (store address + (2x4)) into $ra
    - Jump to the second instruction after the branch (see page 139 of MIPS32™ Architecure For Programmers Volume II, Revision 2.00)
  - jump to address addr
- jr $ra
  - allows subroutine to jump back
    - Execute the instruction following the jump, in the branch delay slot, before jumping (page 145 of document above)
  - care must be taken to preserve $ra!
  - more work for non-leaf procedures

(No) delayed branches



- PCPSIM: Simulator➔settings
  - If these options have "no tick" then jump is to next address (also the next address is not executed in case of a branch).

Procedures

- one of the few means to structure your assembly language program
- small entities that can be tested separately
- can make an assembly program more readable
- recursive procedures

Write your own procedures

```
# prints the integer contained in $a0
print_int:
        li $v0, 1           # system call to
        syscall             # print integer
        jr $ra              # return
        nop                 # this instruction is executed
                            # required in case of a
                            # 'delayed' branch.
main: . . .
        li $a0, 10          # we want to print 10
        jal print_int       # print integer in $a0
```

Write your own procedures

```
        .data
linebrk: .asciiz "\n"
        .text
print_eol:                  # prints "\n"
        li $v0, 4           #
        la $a0, linebrk     #
        syscall             #
        jr $ra              # return
        nop                 # this instruction is executed
                            # required in case of a
                            # 'delayed' branch.
main: . . .
        jal print_eol       # printf("\n")
```

Write your own procedures

```
        .data
main:
        li $s0, 1           # $s0 = loop ctr
        li $s1, 10          # $s1 = upperbnd
loop:   move $a0, $s0       # print loop ctr
        jal print_int       #
        jal print_eol       # print "\n"
        addi $s0, $s0, 1    # loop ctr +1
        ble $s0, $s1, loop  # unless $s0>$s1…
```

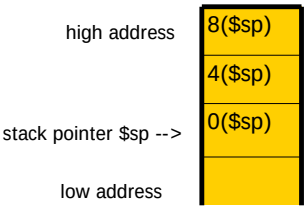
Non-leaf procedures

- Suppose that a procedure procA calls another procedure jal procB
- **Problem:** jal stores return address of procedure procB and destroys return address of procedure procA
- Save $ra and all necessary variables onto the stack, call procB, and restore

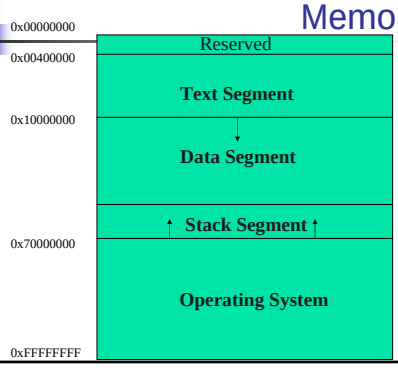
Stack

The stack can be used for

- parameter passing
- storing return addresses
- storing result variables
- stack pointer $sp



Memory layout



Factorial

0! = 1

1! = 1

N! = N × (N-1)!

Factorial is a recursive procedure with one argument $v0

Solution based on description in "MIPS Assembly Language programming"

```
            .data
prompt:     .asciiz  "\n\n Give me a value for 'N': "
msg:        .asciiz  " N factorial is: "
bye:        .asciiz  "\n### Good-bye ###"
            .text
.globl      main
main:       addiu    $sp, $sp, -8          # allocate space
mloop:
            li       $v0, 4                                  } Print prompt
            la       $a0, prompt                             }
            syscall                                          }
            li       $v0, 5                # Get value for N } Get N
            syscall                                          }
            bltz     $v0, quit                               } Quit if <0
            sw       $v0, 0($sp)
            jal      fac                   # Call factorial
            nop
            li       $v0, 4                # Print message
            la       $a0, msg
            syscall
            lw       $a0, 4($sp)           # Get result
            li       $v0, 1
            syscall
            b        mloop
quit:
            addiu    $sp, $sp, 8# Deallocate space
            li       $v0, 4
            la       $a0, bye
            syscall
            li       $v0, 10
            syscall
```

```
fac:
            lw       $a0, 0($sp)                                  } Check if value is
            bltz     $a0, problem                                 } Between 0 and 13
            addi     $t1, $a0, -13                                }
            bgtz     $t1, problem    # 13 is largest value we can accept
            addiu    $sp, $sp, -16   # Allocate
            sw       $ra, 12($sp)    # Save return address
            sw       $a0, 8($sp)
            slti     $t0, $a0, 2     # If N is 1 or 0 then return the value 1  } If value is 0 or 1
            beqz     $t0, go                                                   } no recursive call
            li       $v0, 1                                                    }
            b        facret                                                    }
```

```
go:         addi     $a0, $a0, -1
            sw       $a0, 0($sp)          # Pass N-1 to factorial function
            jal      fac                  # Recursive call
            nop
            lw       $v0, 4($sp)          # Get (N-1)! back
            lw       $ra, 12($sp)
            lw       $a0, 8($sp)
            mult     $v0, $a0  # N*(N-1)!
            mflo     $v0
facret:     addiu    $sp, $sp, 16         # Deallocate
            sw       $v0, 4($sp)
            jr       $ra
problem: sw      $v0, 4($sp)
            jr       $ra
```