MIPS

1 Introduction

MIPS - Microprocessor without Interlocked Pipeline Stages

Underlying design principles:

- Simplicity favours regularity
- Make the common case fast
- Smaller is faster
- Good design demands good compromises

2 32 Bit RISC Processor

- Around 80 instructions in the instruction set
- 32 general purpose registers $r0 - $r31
- $r0 is special and always contains the value 0
- The MIPS processor has a super pipelined architecture - each instruction is broken down into a sequence of 'micro' instructions

3 Design principles

MIPS is a reduced instruction set computer (RISC), with a small number of simple instructions. Other architectures, such as intel's x86 are complex instruction set computers (CISC)

- Simplicity favours regularity
  - Consistent instruction format: same number of operands (two sources and one destination) is easier to encode and handle in hardware
- Make the common use case fast
  - MIPS includes only simple, commonly used instructions
  - Hardware to decode and execute instructions can be simple, small and fast
  - More complex instructions (that are less common) performed using multiple simple instructions
- Smaller is faster
  - MIPS includes only a small number of registers

Name	Register No.	Usage
$0	0	the constant value 0
$at	1	assembler temporary
$v0-$v1	2-3	Function return values
$a0-$a3	4-7	Function arguments
$t0-$t7	8-15	temporaries
$s0-$s7	16-23	saved variables
$t8-$t9	24-25	more temporaries
$k0-$k1	26-27	OS temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	Function return address

- Good design demands good compromises
  - Multiple instruction formates allow flexibility, for example some use 3 operands, some 2
  - Number of instruction formats kept small to adhere to design principles 1 and 3
  - Other formats appear in assembler, but are transformed into machine code to fit with this format

4 Instruction Types

4.1 R Type

R-Type

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

- 3 register operands:
  - `rs, rt`: source registers
  - `rd`: destination register
- Other fields:
  - `op`: the operation code or **opcode** (0 for R-type instructions)
  - `funct`: the **function,** with opcode, tells computer what operation to perform
  - `shamt`: the **shift amount** for shift instructions, otherwise it's 0

Examples:

`add $s0, $s1, $s2`
"add values in registers 17 and 18 and put the answer in register 16."

`sub $t0, $t3, $t5`
"subtract values in registers 13 from 11 and put the answer in register 8."

Field Values

op	rs	rt	rd	shamt	funct
0	17	18	16	0	32
0	11	13	8	0	34
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Machine Code

op	rs	rt	rd	shamt	funct	
000000	10001	10010	10000	00000	100000	(0x02328020)
000000	01011	01101	01000	00000	100010	(0x016D4022)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Note: assembler and machine code order the operands differently

Examples:

`sll $s0, $s1, 5`

"shift bits in register 17 left 5 places and put in in register 16."

Field values:

op	rs	rt	rd	shamt	funct
0	0	17	16	5	0

Machine code:

000000	00000	10001	10000	00101	00000

Note: assembler and machine code order the operands differently

4.2 I Type

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

- 3 operands:
  - `rs`, `rt`: register operands
  - `imm`: 16-bit two's complement immediate
- Other fields:
  - `op`: the **opcode**, operation is completely determined by opcode

Examples:

`addi $s0, $s1, 5`

"add value in register 17 and '5' and put the answer in register 16."

Field values:

op	rs	rt	imm
8	17	16	5

Machine code:

001000	10001	10000	0000 0000 0000 0101

4.3 J Type

J-Type

op	addr
6 bits	26 bits

- 26-bit address operand: `addr`
- Used for jump instructions: `j`

- Rarely used in assembler
- Typically use the R-type instruction: `jr name` or `jr $s0`
  (Jump to name, or jump to address contained in register)

5 Addressing

How do we address the operands?

- **Register Only** — `add $s0, $s1, $s2`
- **Immediate** (16-bit two's complement integer) — `addi $s0, $s1, 5` / `ori  $t3, $t7, 0xFF`
- **Base Addressing** *address of* operand is given by base address + signed immediate — `lw $s0, 0($sp)` / `sw $s0, -12($t0)`
- **PC-Relative** jump so far from current position — `beq  $t0, $0, else` becomes `beq  $t0, $0, 3`

There is the 3 at the end of the pc relative jump because it jumps 3 lines ahead in the program

Obtaining memory addresses:

- declare data at the beginning of the program and look up addresses of variables.

```
.data
string1: .space 10
string2: .asciiz "Oh:"
var1: .word 1234

.text
.globl main
main:
lw      $s0, var1
la      $a0, string1
```

6 Loading 32 Bit words

How, if you only have 16-bit immediates?

- **Answer:** load first 16-bits with special command, then stick on the rest
- E.g. want to add `0xFEDC8765`
- First add `0xFEDC0000` then add `0x00008765`

```
lui $s0, 0xFEDC
ori $s0, $s0, 0x8765
```

This loads half into the left half of the bits, and the other half into the right half

7 OS Calls

Set call type in register `$v0`, e.g. `ori $v0, $0, 10`
Use assembly code: `syscall`

Service	Code(in $v0)	Arguments / Results
print_int	1	`$a0` = integer to be printed
print_float	2	`$f12` = float to be printed
print_double	3	`$f12` = double to be printed
print_string	4	`$a0` = address of string in memory
read_int	5	integer returned in `$v0`
read_float	6	float returned in `$v0`
read_double	7	double returned in `$v0`
read_string	8	`$a0` = memory address of string input buffer `$a1` = length of string buffer (n)
sbrk	9	`$a0` = amount, address in `$v0`
exit	10	

8 Multiplication and Division

32 × 32 multiplication, 64 bit result
- `mult $s0, $s1`
- Result in special registers: `lo, hi`

32-bit division, 32-bit quotient, remainder
- `div $s0, $s1`
- Quotient in `lo`
- Remainder in `hi`

Moves from `lo/hi` special registers
- `mflo $s2`
- `mfhi $s3`

Note: We used `mul` earlier.
This gives a 32-bit result
with no overflow checking

```
mul $v2,$s3,$t0
```
Translates into:
```
mult  $s3, $t0
mflo  $v2
```

9 MIPS Function Calls

```
main:
li $s1, 123
li $s2, 234
li $t3, 456
li $t5, 345
jal adder
ori $v0, $0, 10
syscall
```

Jump and Link – stores return address in `$ra`

```
adder:
add $s0, $s1, $s2
jr $ra
.end
```

Jump Register – jumps to address in `$ra`

10 Conventions

Caller:

- passes arguments to callee – using registers `$a0-$a4`
- jumps to callee – using `jal`

Callee:

- performs the function
- returns result to caller – using registers `$v0-$v1`
- returns to point of call – using `jr` to `$ra`
- must not overwrite registers or memory needed by caller
  i.e. $s0-$s7, $ra, $sp
  Have to save these values on the stack if using the registers

Note: convention assumes can write over $t registers

The Stack:

- a dynamically sized chunk of memory
- $sp always contains the address of the head of the stack

To add to the stack:

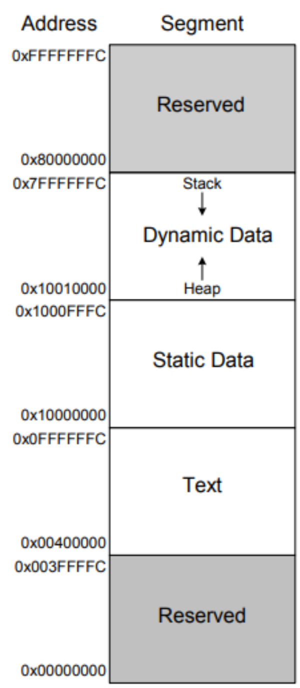
- move the stack pointer down one pos^{n}.
- write the value

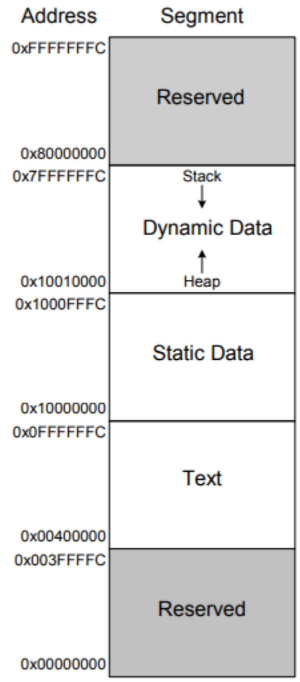
```
addi $sp, $sp, -4
sw   $s0, 0($sp)
```

To pop from the stack:

- read the value
- move the stack pointer up one pos^{n}.

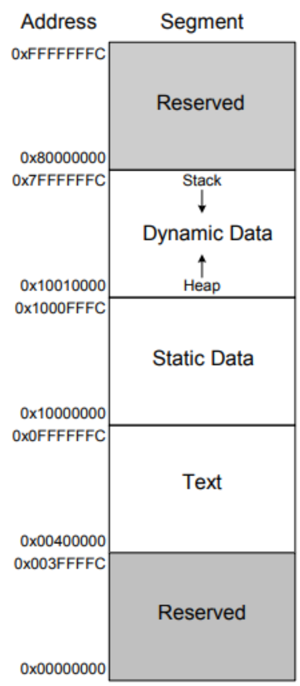
```
addi $sp, $sp, 4
lw   $s0, 0($sp)
```





Recursive call:

- must preserve `$ra` so that prior call can return to the correct place
- so store it on stack before calling a function
- reinstate it afterwards



Caller

- Put arguments in `$a0-$a3`
- Save any needed registers (`$ra`, maybe `$t0-t9`)
- `jal` callee
- Restore registers
- Look for result in `$v0`

Callee

- Save registers that might be disturbed (`$s0-$s7`)
- Perform function
- Put result in `$v0`
- Restore registers
- `jr $ra`

11 Example - Factorials

```
factorial:
   addi $sp, $sp, -4   # make room
   sw   $a0, 0($sp)    # store $a0
   addi $sp, $sp, -4   # make room
   sw   $ra, 0($sp)    # store $ra
   addi $t0, $0, 2
   slt  $t0, $a0, $t0  # if $a0<2, set $t0 to 1, otherwise 0
   beq  $t0, $0, else  # $a0 wasn't <2: go to else
   addi $v0, $0, 1     # $a0 was <2: return 1
   addi $sp, $sp, 8    # restore $sp
   jr   $ra            # return
else: addi $a0, $a0, -1   # n = n - 1
   jal  factorial      # recursive call
   lw   $ra, 0($sp)    # restore $ra
   lw   $a0, 4($sp)    # restore $a0
   addi $sp, $sp, 8    # restore $sp
   mul  $v0, $a0, $v0  # n * factorial(n-1)
   jr   $ra            # return
```