MIPS assembler Language

Processor-Programmer communication

Assembler language.

1 instruction corresponds to 1 bit string sent to the processor and interpreted as an instruction.

1 instruction (mnemonic) corresponds to a primitive action understood by the programmer (compiler) writer.

Assembler and ISA are developed together
the architect has knowledge of both

We need to choose.
I have chosen to start from the programmers view.
Your knowledge of computing will help you to understand the language.

When we know what is to go implemented we look at how it is implemented.

Registers

Computation at the basic level is arithmetic.

Most modern computer architectures do not allow arithmetic operations on the values in memory.

Arithmetic is done performed on values in ***Registers***

We shall see that optimisation and pipelining benefits from many registers.

However another problem limits the practical number of registers.

Early days it was partially cost.
Now more mundane.

We need to address registers.

Registers

2 registers can be identified by 1 bit
4 registers can be identified by 2 bits
8 registers can be identified by 3 bits

2^n registers can be identified by n bits

MIPS architecture has 32 registers – 5 bits

Suppose we want to move data from memory to a register.

The bit pattern in memory must include a part for the instruction; a part for the register address; a part for the memory address.

128 instructions needs 7 bits.
So address has 20 bits for a 32 bit machine.

2^{20} is 1 million – so can only address 4 Megabytes of memory directly. A problem

Byte addresses not bit addresses

Registers Arithmetic

Register 1 = Register 2 + Register 3

Three register addresses (15 bits) + instruction (7 bits) = 22 bits

Load / Store

Moving a word to and from memory is known as "loading" and "storing.

lw $t0, 8($s1) Load word

$t0 is the destination register

8($s1) is the memory address where the data needs to come from

sw $t0, 8($s1) Store word

reverse procedure

Offset is number of words

$s1 is a register which contains an address in memory and 8 is the offset from that address where the data can be found.

Memory addressing

$s1 is a 32 bit register and by getting the address of the memory location from there we can access 2^{32} locations.
Or about 4 Gbytes, which is where the address space for 32 bit machines comes from.

$s1 is the base address
8 is the offset.

Allowing offsets like this allows us to easily loop through a number of locations.

Our example

The MIPS architecture also allows the loading (and storing) of bytes and half words.

Addition

With data in registers manipulation Is possible.

```
add  $t0, $s1, $s2
```

puts the result of adding the values in $s1 to that in $s2 and storing the result in $t0.

```
sub  $t0, $s1, $s2
```

compilation

```
a = (b + c) – (d + f)                    Java
```

load instructions b -> f into $s0 to $s3

```
add $t0, $s0, $s1                     b+c
add $t1, $s2, $s3                     d+f
sub $s4, $t0, $t1                 (b+c) –(d+f)
```

store $s4 back into memory location for a

Alternative

```
add $s4, $s0, $s1                 b+c
sub $s4, $s4, $s2                 b+c-d
sub $s4, $s4, $s3                 (b+c) –(d+f)
```

Saved on registers – need to know rules of arithmetic.

Compiler intelligence v. register number.

The performance of a computer system depends on the hardware AND software.

The optimisations which can be performed by the hardware and software to some extent overlap.

Hardware improvements measured on unoptimised code are unlikely to be reproduced on optimised code.

Optimisations are not (necessarily) independent.

Instruction fields

An R-type MIPS instruction has the following structure.

op code	– 6bits	command
rs	- 5bits	source register 1
rt	- 5bits	source register 2
rd	- 5bits	destination register
shamt	- 5bits	shift amount
funct	- 5bits	function code – selects variant of opcode

Very simple instruction format. Many machines have far more complex instruction formats.

A constant format has advantages when it comes to performance.

In particular when we are trying to implement *instruction level parallelism.*

Binary codes

op code – 6bits	command
rs - 5bits	source register 1
rt - 5bits	source register 2
rd - 5bits	destination register
shamt - 5bits	shift amount
funct - 5bits	function code – selects variant of opcode

add $s1, $s2, $s0

```
op code = 000000
funct   = 100000
Shamt   = 000000
rs      =  01001
rt      =  01010
rd      =  01000
```

000000 **0**10001 **0**1010 **0**1000 **1**00000

Inherent Parallelism

What happens when an instruction is executed. Say an **add**

Instruction is fetched from memory (cache).

Instruction is decoded – it is an add ← Decode 1

Source register 1 decoded
Data moved from b to $s1 ← Decode 2
Source register 2 decoded
Data moved from c to $s2

Memory access

If we have a single decoder

Sum stored in $s0

Destination register decoded
Data moved from $s0 to a

Memory access

If we can overlap parts of the instruction it can run more quickly (without increasing the clock frequency

12 I-type

Complications

Even the MIPS dataset cannot work with just one instruction format – the R-type.

I-type

op code	– 6bits	command
rs	- 5bits	source register 1
rt	- 5bits	source register 2
	- 16bits	constant or address

We can add a constant to a register by using the **add immediate** instruction

```
addi  $t0, $t0, 25
```

Adds 25 to the value in $t0 and stores in $t0

Note there is no subtract immediate – you use a negative constant.

Reduces the range … reduces the number of instructions

Logical Ops

	Java	MIPS
Shift left	<<	sll
Shift right	>>	srl
Bitwise and	&	and, andi
Bitwise or	|	or, ori
Bitwise not	~	not

No integer multiply – just use shift and add

Branch Ops

Conditional branches causes problems in piplining. Next instruction is not known

```
beq  $reg1, $reg2, Label
bne  $reg1, $reg2, Label
```

Conditional branch

Jumps to the label is the registers are (not) equal.

```
j label
```

Jumps straight to label

Unconditional branch

Set Ops

```
slt $s1, $s2, $s3          if $s2<$s3 $s1=1,
                           else $s=0
slti $s1, $s2, 100
```

No branch on less than equality.
Need to do set a value and then jump on a value.

Why no branch on less than?

Number of instructions?
Simpler instructions …

Fight between instruction complexity and clock speed.

1 more instruction for this sort of branch against slower clock for **ALL** instructions.

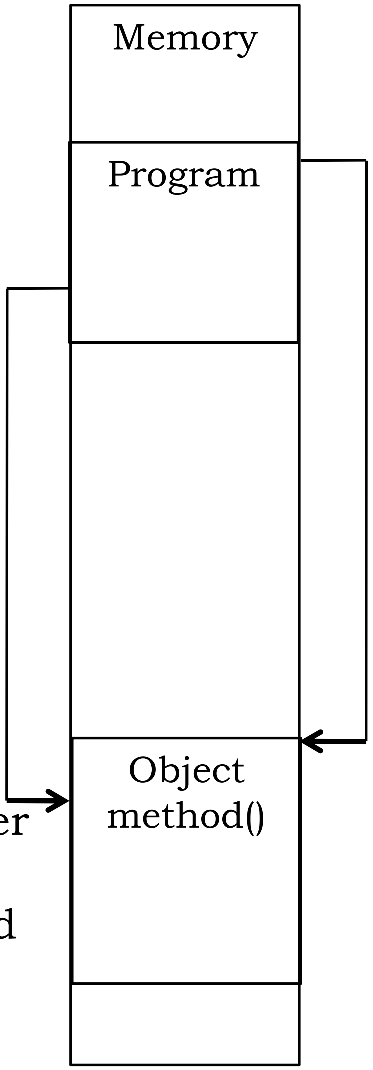
Make the common case fast

Problems

Useful construct

Program
…
…
Message to method
….
Message to method

Object
…
method()
….
return

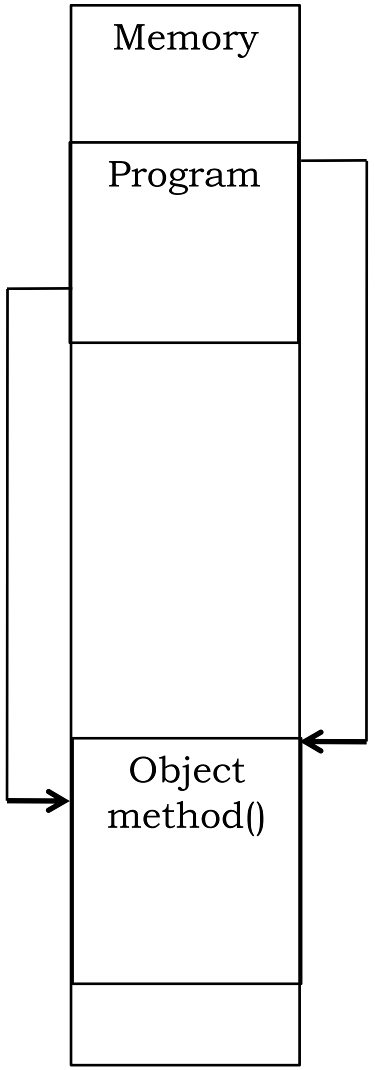


Knowing where to move to is simple from the loader

How to get back at the end of the method?
How to transfer data?
How to return data?
Existing register values permanent

Message to method equivalent call procedure

Starting address of symbolic name recorded



Architectural support

To send a message to an object.

1. Place parameters in a place for the procedure
2. Transfer control to the procedure
3. Acquire storage resource
4. Perform method
5. Place result for program to pick up after method
6. Return control to the point of origin

Message to method equivalent call procedure

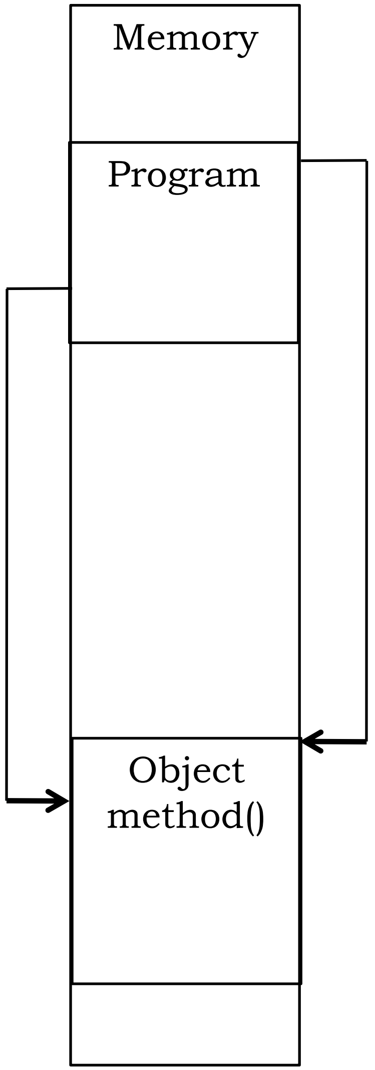
Common problem – solution: special registers for data transfer; special commands to execute the necessary steps.
VAX did a lot automatically
MIPS takes a less comprehensive approach.

Recursion makes it worse

MIPS is easier to tailor to needs.

Best place to hold data for moving back and forward is registers:

fast;
in a well known place;



Architectural support

MIPS conventions

\$a0-\$a4	argument registers parameter passing.
\$v0-\$v1	value registers for return
\$ra	return address register
jal <method>	jump and link places the PC+4 in \$ra

MIPS provides

jr \$ra	jump register. Unconditional jump to all the address space call at end of the method

Also used for conditional jumps to distant places.

bneq condition <target>
    branch not equals – limited range. (bits for instruction)

```
beq    condition  +2
jr     $reg
```
invert condition and unconditional jump

Compiler support

Stack
filled

Registers and stack

Registers in use for the programme, now needed for method

Spill the registers put them into memory for later retrieval. (May not have memory locations – intermediate values)
For recursive calls – which value goes back into memory?

Stack *last in – first out queue*
Best way to spill registers.

Stack pointer
Pointer to most recently allocated address.

Push: increment pointer and add value to stack
Pop: take value from stack and decrement

Conventional a stack is placed at the top of memory and filled down. Using pop and push allows us to ignore actual direction.
Always refer to “incrementing stack pointer.
MIPS software defines $sp – stack pointer

Compiler support

Stack

Registers and stack

```
addi   $sp, $sp, -12           grow stack
sw     $s2, 8($sp)             push items
sw     $s1, 4($sp)
sw     $s0, 0($sp)
```

Also convention
$t registers are not required by the procedure and may be used.
$s registers are required and must be preserved by the method call

target must only add to the stack and remove what it put there.

Higher regions of stack must be undisturbed.

target which calls another method must push the $ra onto the stack before making a call.

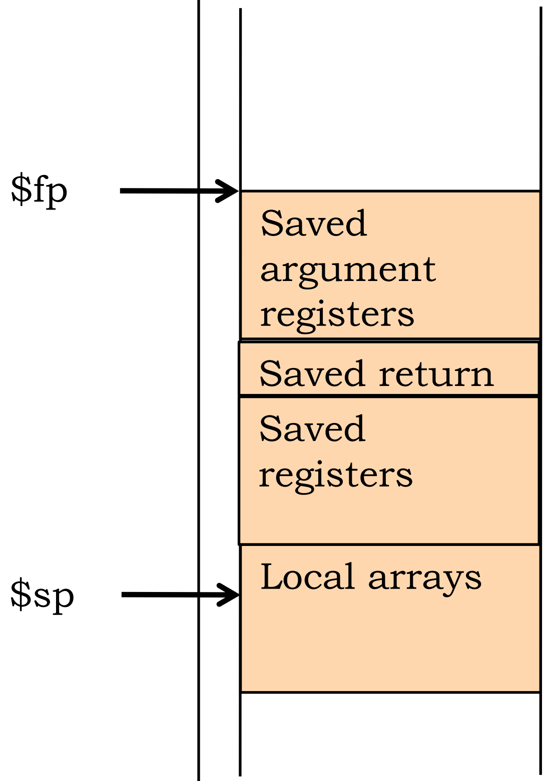
Stack is also used for variables local to the code such as local arrays.

Frame pointer

The stack pointer changes during a procedure.

Address of variables (offsets from $sp) change if the $sp changes. Creates problems!

Define the **frame point *$fp*** – this points to the start of the stack



The portion which contains the methods saved registers and local data is called the **Procedure frame** or **Activation record**

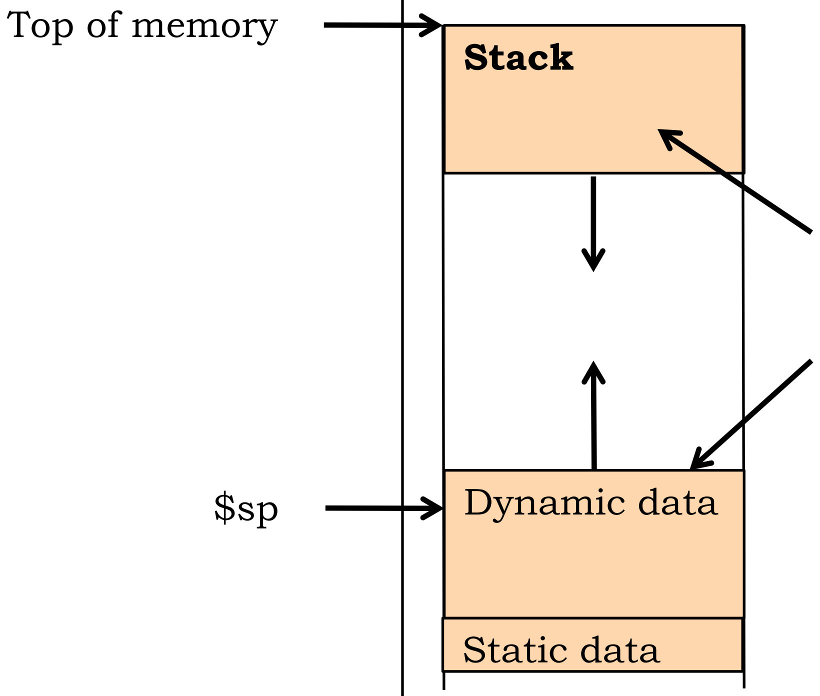
Start to understand the java stack trace utility.

Progress through a programme is stored on the stack. *Popping* the stack is equivalent to *unwinding* the calling sequence

Heap

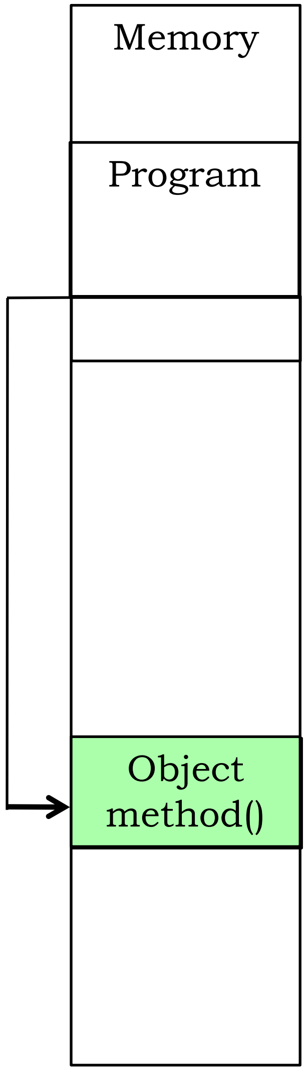
Automatic variables are local to a method (procedure). When the segment exits the values disappear.

Some values should remain between invocations *Static* data – this is placed immediately after the code.
Some structures are also needed across invocations but their size varies – *dynamic* data placed after the static data



Stack grows down
Heap grows up

Global Pointer $gp is a convention (not an architectural decision)
Initialised to "the middle of the heap" – so we can access the maximum range uses offsets from $gp



Compiler support

In-lining

In order to avoid all this overhead (for small pieces of code) in-line.

Move the code to the place it is wanted.
By-pass all the complex code, at the cost of memory space.

C programmer can specify code to be “in-lined”

Object-Oriented

Each object needs a new area of memory
Need to be able to copy code/data during the programme execution not just linking.

Methods (procedures) which only take a few lines are expensive.

At one time told don’t break them out.
Now rely on in-lining, but keep code clear to allow optimisation

Register spills

$sp: the stack pointer register.
Points to a place in memory whither you can spill the registers. Stack is at the top of memory and grows down.

```
addi $sp, $sp -16                four words
sw $t0, 12($sp)
sw $t1, 8($sp)
sw $t2, 4($sp)
sw $t3, 0($sp)
```

call

Must decrement the stack pointer so calling routine can use stack.

return

```
lw $t3, 0($sp)
lw $t2, 4($sp)
lw $t1, 8($sp)
lw $t0, 12($sp)
addi $sp, $sp, 16
```

If the procedure makes a call itself if must store $ra before, and restore it after

Conventional name/uses

Name	Number	Use
$zero	0	Constant 0
$at	1	Assembler temporary
$v0-$v1	2-3	Function results
$a0-$a4	4-7	Arguments
$t0-$t7	8-15	Temporaries
$s0-$s7	16-23	Saved temporaries
$t8-$t9	24-25	Temporaries
$k0-$k1	26-27	Reserved for OS kernel
$gp	28	Global pointer
$sp	29	Stack pointer
$fp	30	Frame pointer
$ra	31	Return address

Method Actions

For a method which makes no extra calls

```
addi $sp, $sp, -4*n        Expand stack
sw   $s0, 0($sp)           Store variable
sw   $s1, 4($sp)           Store variable
code
lw   $s1, 4($sp)           Restore variable
lw   $s0, 0($sp)           Restore variable
addi $sp, $sp, 4*n         Shrink stack
jr   $ra
```

For a method which makes **extra** calls

```
addi $sp, $sp, -4*n        Expand stack
sw   $ra, 0($sp)           Save return
sw   $fp, 4($sp)           Save frame
sw   $a0, 8($sp)           Save any args
sw   $t0, 12($sp)          Save any temps
code
lw   $t0, 12($sp)          Restore temps
lw   $a0, 8($sp)           Restore args
lw   $fp, 4($sp)           Restore frame
lw   $ra, 0($sp)           Restore return
addi $sp, $sp, 4*n         Shrink stack
jr   $ra                   Return
```

Byte transfer

To handle text which is stored in bytes, the MIPS has two more instructions

lb $t0, 0($sp)	load byte
sb $t0, 0($sp)	load byte

Hence the address is byte not word.

32 bit addresses.

It is useful to be able to set all 32 bits of a register. This is not possible with the commands given because the addi only has 16 bits for the data.

lui $t0, 245	load upper immediate, loads 245 into the upper byte of $t0
ori $t0, $t0, 312	will then put 312 into the lower byte of $t0

J-Format

The jump format has a final word format

op	6 bits
address	26 bits

It allows the maximum length jump without any further calculation.

We will see that it is possible to decode bits 7-22 as if they were register addresses and to just ignore their values for a jump instruction.

We do not wish to have to decode the op code in order to find out what the various bits of the command mean.

MIPS addressing - review

We wish to designate points in memory both to load/store data and to set the **PC (**programme counter).

Register: address in the register

Base/Displacement: address is the contents of a register plus a constant.

Immediate: value in instruction

PC relative: current position +/- a constant

Pseudodirect: upper 6 bits of the PC concatenated with the 26 bits of the jump address

Other modes are used in other machines and we have looked at some of those.

PC is a hardware location which contains the next address to be executed

29 Pseudo-insrtuctions

Pseudo-instructions

Instructions which are recognised by the assembler and are translated into machine code, but are not implemented by the hardware.

move
mult
multi
li
div

The assembler will translate them into a number real instructions which are implemented in hardware.

Makes writing compilers for the hardware easier.