Computer Architecture CSL3020

Deepak Mishra

http://home.iitj.ac.in/∼dmishra/
Department of Computer Science and Engineering
Indian Institute of Technology Jodhpur

ISA

ISA is an abstract interface between hardware and software that encompasses all the information necessary to write a machine language program.

Example of instructions in an ISA

- Arithmetic instructions: add, sub, mul, div
- Logical instructions: and, or, not
- Data transfer/movement instructions

ISA Features

Following features are desirable in an ISA

Slide credit: SR Sarangi

ISA Features

Following features are desirable in an ISA

- **Completeness**: It should be able to implement all the programs that users may write.

Slide credit: SR Sarangi

ISA Features

Following features are desirable in an ISA

- **Completeness**: It should be able to implement all the programs that users may write.
- **Conciseness**: The instruction set should have a limited size. Typically an ISA contains 32-1000 instructions.

Slide credit: SR Sarangi

ISA Features

Following features are desirable in an ISA

- **Completeness**: It should be able to implement all the programs that users may write.
- **Conciseness**: The instruction set should have a limited size. Typically an ISA contains 32-1000 instructions.
- **Generic**: Instructions should not be too specialized, e.g. add14 (adds a number with 14) instruction is too specialized.

Slide credit: SR Sarangi

ISA Features

Following features are desirable in an ISA

- **Completeness**: It should be able to implement all the programs that users may write.
- **Conciseness**: The instruction set should have a limited size. Typically an ISA contains 32-1000 instructions.
- **Generic**: Instructions should not be too specialized, e.g. add14 (adds a number with 14) instruction is too specialized.
- **Simplicity**: Should not be very complicated.

Slide credit: SR Sarangi

ISA Paradigms

There are two popular ISA paradigms

ISA Paradigms

There are two popular ISA paradigms

- **RISC**: Reduced Instruction Set Computer.

ISA Paradigms

There are two popular ISA paradigms

- **RISC**: Reduced Instruction Set Computer.

  RISC implements simple instructions that have a simple and regular structure. The number of instructions is typically a small number e.g. ARM.

ISA Paradigms

There are two popular ISA paradigms

- **RISC**: Reduced Instruction Set Computer.

  RISC implements simple instructions that have a simple and regular structure. The number of instructions is typically a small number e.g. ARM.

- **CISC**: Complex Instruction Set Computer.

ISA Paradigms

There are two popular ISA paradigms

- **RISC**: Reduced Instruction Set Computer.

  RISC implements simple instructions that have a simple and regular structure. The number of instructions is typically a small number e.g. ARM.

- **CISC**: Complex Instruction Set Computer.

  CISC implements complex instructions that are irregular, take multiple operands, and implement complex functionalities. The number of instructions is large e.g. Intel x86

ISA Example

Single instruction ISA

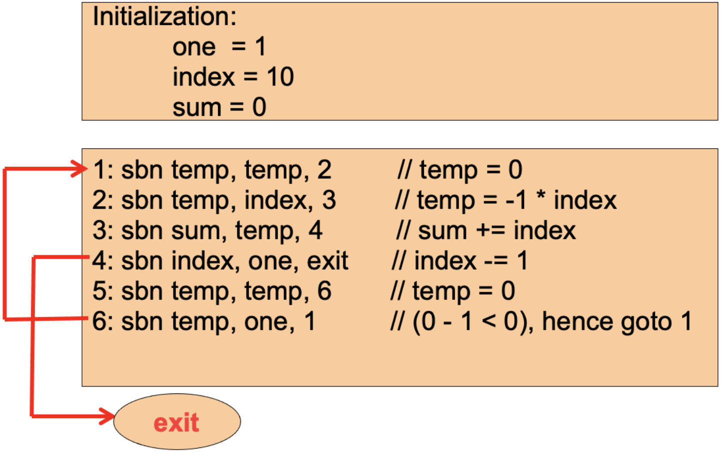
sbn – subtract and branch if negative

Add (a + b) (assume temp = 0)

1. sbn temp, b, 2
2. sbn a, temp, exit

ISA Example

Add the numbers – 1 . . . 10

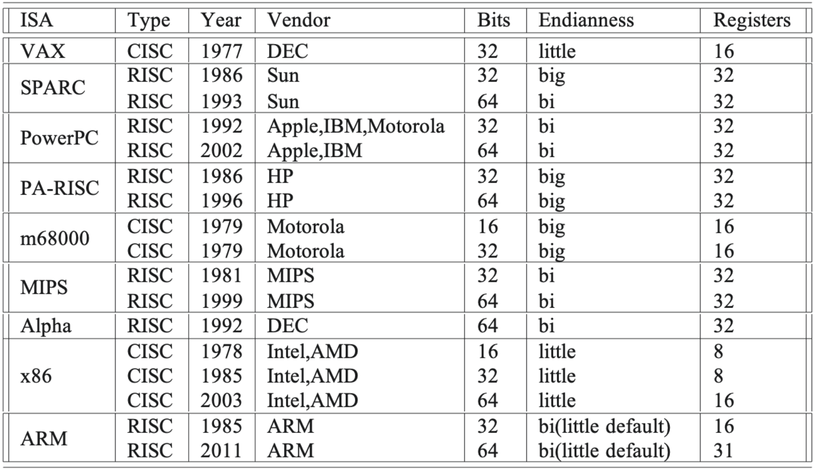


```
Initialization:
        one  = 1
        index = 10
        sum = 0
```

```
1: sbn temp, temp, 2          // temp = 0
2: sbn temp, index, 3         // temp = -1 * index
3: sbn sum, temp, 4           // sum += index
4: sbn index, one, exit       // index -= 1
5: sbn temp, temp, 6          // temp = 0
6: sbn temp, one, 1           // (0 - 1 < 0), hence goto 1
```

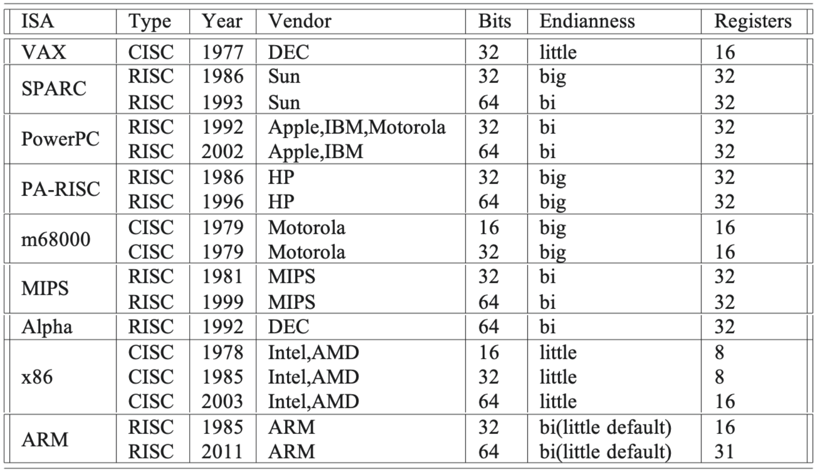
Slide credit: SR Sarangi

Real-world ISA



ISA	Type	Year	Vendor	Bits	Endianness	Registers
VAX	CISC	1977	DEC	32	little	16
SPARC	RISC	1986	Sun	32	big	32
	RISC	1993	Sun	64	bi	32
PowerPC	RISC	1992	Apple,IBM,Motorola	32	bi	32
	RISC	2002	Apple,IBM	64	bi	32
PA-RISC	RISC	1986	HP	32	big	32
	RISC	1996	HP	64	big	32
m68000	CISC	1979	Motorola	16	big	16
	CISC	1979	Motorola	32	big	16
MIPS	RISC	1981	MIPS	32	bi	32
	RISC	1999	MIPS	64	bi	32
Alpha	RISC	1992	DEC	64	bi	32
x86	CISC	1978	Intel,AMD	16	little	8
	CISC	1985	Intel,AMD	32	little	8
	CISC	2003	Intel,AMD	64	little	16
ARM	RISC	1985	ARM	32	bi(little default)	16
	RISC	2011	ARM	64	bi(little default)	31

Real-world ISA



ISA	Type	Year	Vendor	Bits	Endianness	Registers
VAX	CISC	1977	DEC	32	little	16
SPARC	RISC	1986	Sun	32	big	32
	RISC	1993	Sun	64	bi	32
PowerPC	RISC	1992	Apple,IBM,Motorola	32	bi	32
	RISC	2002	Apple,IBM	64	bi	32
PA-RISC	RISC	1986	HP	32	big	32
	RISC	1996	HP	64	big	32
m68000	CISC	1979	Motorola	16	big	16
	CISC	1979	Motorola	32	big	16
MIPS	RISC	1981	MIPS	32	bi	32
	RISC	1999	MIPS	64	bi	32
Alpha	RISC	1992	DEC	64	bi	32
x86	CISC	1978	Intel,AMD	16	little	8
	CISC	1985	Intel,AMD	32	little	8
	CISC	2003	Intel,AMD	64	little	16
ARM	RISC	1985	ARM	32	bi(little default)	16
	RISC	2011	ARM	64	bi(little default)	31

We will consider MIPS instruction set to learn about Assembly Language.

MIPS Instructions

MIPS: Microprocessor without Interlocked Pipeline Stages : ISA
MIPS: Millions Instructions Per Sec: Measure

MIPS Arithmetic

Addition

C code: a = b + c

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

MIPS Arithmetic

Addition

C code: a = b + c

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

Subtraction

C code: a = b - c

MIPS code: sub $s0, $s1, $s2 # register $s0 contains b - c

MIPS Arithmetic

Addition

C code: a = b + c

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

Subtraction

C code: a = b - c

MIPS code: sub $s0, $s1, $s2 # register $s0 contains b - c

Simplicity favors regularity

MIPS Arithmetic

C code: a = (b + c) - (d + e)

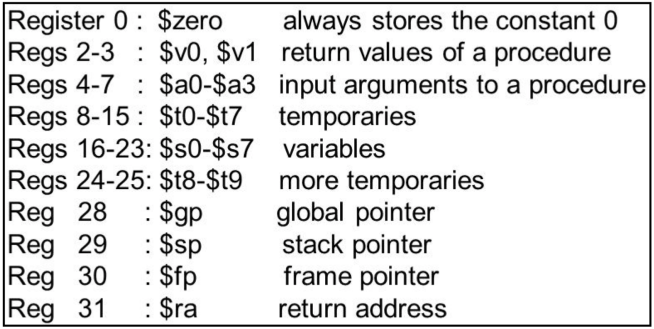
MIPS Arithmetic

C code: a = (b + c) - (d + e)

MIPS code:

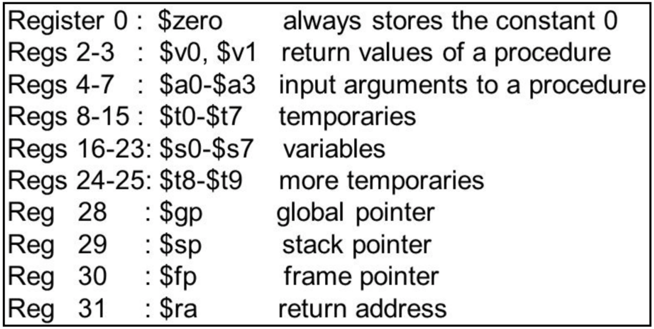
```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

MIPS Registers



Register 0 :	$zero	always stores the constant 0
Regs 2-3 :	$v0, $v1	return values of a procedure
Regs 4-7 :	$a0-$a3	input arguments to a procedure
Regs 8-15 :	$t0-$t7	temporaries
Regs 16-23:	$s0-$s7	variables
Regs 24-25:	$t8-$t9	more temporaries
Reg 28 :	$gp	global pointer
Reg 29 :	$sp	stack pointer
Reg 30 :	$fp	frame pointer
Reg 31 :	$ra	return address

MIPS Registers



Register 0 :	$zero	always stores the constant 0
Regs 2-3 :	$v0, $v1	return values of a procedure
Regs 4-7 :	$a0-$a3	input arguments to a procedure
Regs 8-15 :	$t0-$t7	temporaries
Regs 16-23:	$s0-$s7	variables
Regs 24-25:	$t8-$t9	more temporaries
Reg 28 :	$gp	global pointer
Reg 29 :	$sp	stack pointer
Reg 30 :	$fp	frame pointer
Reg 31 :	$ra	return address

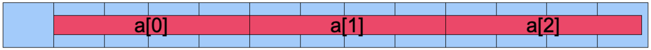
Why are there only 32 registers?

MIPS Arithmetic

C code: A[12] = h + A[8];

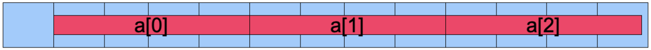
MIPS Arithmetic

C code: $A[12] = h + A[8];$



MIPS Arithmetic

C code: $A[12] = h + A[8];$



Big endian byte order

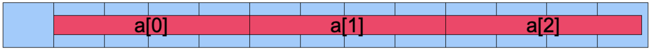
0	1	2	3
4	5	6	7

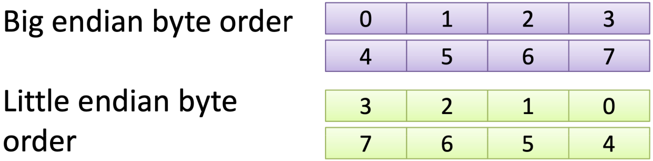
Little endian byte order

3	2	1	0
7	6	5	4

MIPS Arithmetic

C code: $A[12] = h + A[8];$





MIPS code:

```
lw $t0, 32($s0)
add $t0, $s1, $t0
sw $t0, 48($s0)
```

MIPS Arithmetic

addi $s3, $s3, 4 # add immediate

MIPS Instructions

MIPS Assembly to Machine Instructions

add $t0, $s1, $s2

MIPS Instructions

MIPS Assembly to Machine Instructions

add $t0, $s1, $s2

0	17	18	8	0	32

MIPS Instructions

MIPS Assembly to Machine Instructions

add \$t0, \$s1, \$s2

0	17	18	8	0	32
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

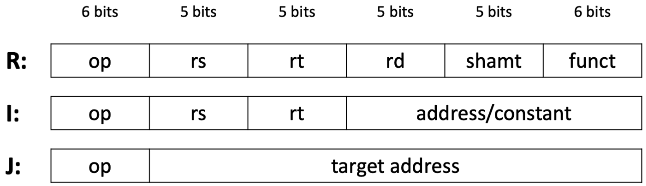
MIPS Instructions

MIPS Assembly to Machine Instructions

add $t0, $s1, $s2

0	17	18	8	0	32
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
000000	10001	10010	01000	00000	100000

MIPS Instructions



MIPS Instructions

	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
R:	op	rs	rt	rd	shamt	funct
I:	op	rs	rt	address/constant		
J:	op	target address				

- op: Basic operation of the instruction (opcode).
- rs: The first source operand register.
- rt: The second source operand register.
- rd: The destination operand register.
- shamt: Shift amount.
- funct: Function. This field selects the specific variant of the operation in the op field.
- address: Offset for load/store instructions
- constant: Constants for immediate instructions

MIPS Instructions

MIPS Assembly to Machine Instructions

lw $t0, 32($s0)

MIPS Instructions

MIPS Assembly to Machine Instructions

lw $t0, 32($s0)

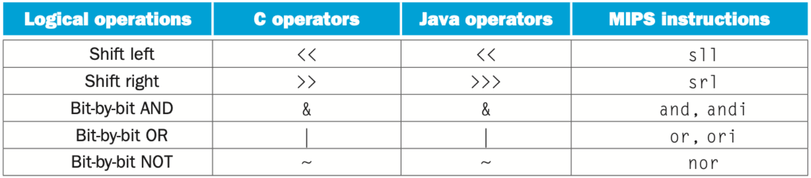
35	16	8	32

MIPS Instructions

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add `$s1,$s2,$s3`
sub	R	0	18	19	17	0	34	sub `$s1,$s2,$s3`
addi	I	8	18	17	100			addi `$s1,$s2,100`
lw	I	35	18	17	100			lw `$s1,100($s2)`
sw	I	43	18	17	100			sw `$s1,100($s2)`
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format

MIPS Logical Instructions

Logical Operations



Logical operations	C operators	Java operators	MIPS instructions
Shift left	`<<`	`<<`	`sll`
Shift right	`>>`	`>>>`	`srl`
Bit-by-bit AND	`&`	`&`	`and, andi`
Bit-by-bit OR	`|`	`|`	`or, ori`
Bit-by-bit NOT	`~`	`~`	`nor`

MIPS Logical Instructions

For example if register \$s0 contains

0000 0000 0000 0000 0000 0000 0000 1001

```
sll $t2, $s0, 4     # reg $t2 = reg $s0 << 4 bits
```

MIPS Logical Instructions

For example if register \$s0 contains

0000 0000 0000 0000 0000 0000 0000 1001

```
sll $t2, $s0, 4     # reg $t2 = reg $s0 << 4 bits
```

The value in register \$t0 would be

0000 0000 0000 0000 0000 0000 1001 0000

srl is similar to sll.

MIPS Logical Instructions

AND operation

For example, if register \$t2 contains
0000 0000 0000 0000 0000 1101 1100 0000

and register \$t1 contains
0000 0000 0000 0000 0011 1100 0000 0000

and \$t0, \$t1, \$t2

MIPS Logical Instructions

AND operation

For example, if register $t2 contains
0000 0000 0000 0000 0000 1101 1100 0000

and register $t1 contains
0000 0000 0000 0000 0011 1100 0000 0000

and $t0, $t1, $t2

The value of register $t0 would be
0000 0000 0000 0000 0000 1100 0000 0000

MIPS Logical Instructions

OR and NOR operations are similar to AND operation

NOT is implemented using NOR as

NOT: A NOR 0 = NOT (A OR 0) = NOT (A)

MIPS also has *and immediate* (andi) and *or immediate* (ori) instructions for performing AND and OR operations with constants

MIPS Branching Instructions

Branching instructions are useful in decision making and implementing loops

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

MIPS Branching Instructions

Branching instructions are useful in decision making and implementing loops

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

```
      bne $s3, $s4, Else
      add $s0, $s1, $s2
      j Exit
Else: sub $s0, $s1, $s2
Exit:
```

MIPS Branching Instructions

```
while (save[i] == k)
i += 1;
```

MIPS Branching Instructions

```
while (save[i] == k)
i += 1;

Loop: sll $t1, $s3, 2     # Temp reg $t1 = i * 4
      add $t1, $t1, $s1   # $t1 = address of save[i]
      lw $t0, 0($t1)      # Temp reg $t0 = save[i]
      bne $t0, $s2, Exit  # go to Exit if save[i] ≠ k
      addi $s3, $s3, 1    # i = i + 1
      j Loop
Exit:
```

MIPS Branching Instructions

Procedure or Functions

Steps for execution of a procedure

1. Put parameters in a place where the procedure can access them.
2. Transfer control to the procedure.
3. Acquire the storage resources needed for the procedure.
4. Perform the desired task.
5. Put the result value in a place where the calling program can access it.
6. Return control to the point of origin.

MIPS Branching Instructions

Procedure or Functions

MIPS has dedicated registers for procedures

- \$a0-\$a3: four argument registers in which to pass parameters
- \$v0-\$v1: two value registers in which to return values
- \$ra: one return address register to return to the point of origin

MIPS Branching Instructions

Procedure or Functions

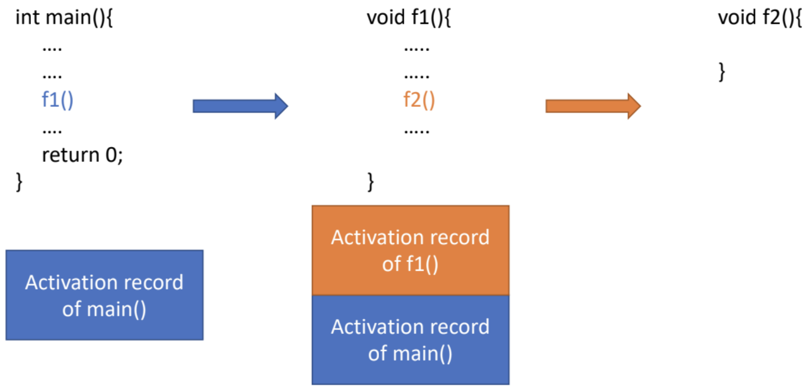
MIPS has dedicated registers for procedures

- \$a0-\$a3: four argument registers in which to pass parameters
- \$v0-\$v1: two value registers in which to return values
- \$ra: one return address register to return to the point of origin

To restore the original values of the registers the stack is used.
- It is accessed through stack pointer register, \$sp (register 29).

MIPS Branching Instructions

Procedure or Functions


int main(){
....
....
f1()
....
return 0;
}
void f1(){
.....
.....
f2()
.....
}
void f2(){
}
Activation record of main()
Activation record of f1()
Activation record of main()

MIPS Branching Instructions

Procedure or Functions

MIPS instruction for calling a procedure is jump-and-link (jal)

```
jal ProcedureAddress
```

jal instruction saves PC + 4 in register $ra. PC (program counter) register holds the address of the current instruction.

To return from a procedure we use

```
jr $ra
```

MIPS Branching Instructions

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

MIPS Branching Instructions

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

Let us assume g, h, i, and j correspond to the argument registers \$a0, \$a1, \$a2, and \$a3, and f corresponds to \$s0.

```
100. leaf: addi $sp, $sp, –4
104.       sw $s0, 0($sp)
108.       add $t0, $a0, $a1
112.       add $t1, $a2, $a3
116.       sub $s0, $t0, $t1
120.       add $v0, $s0, $zero
124.       lw $s0, 0($sp)
128.       addi $sp, $sp, 4
132.       jr $ra
```

MIPS Branching Instructions

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

MIPS Branching Instructions

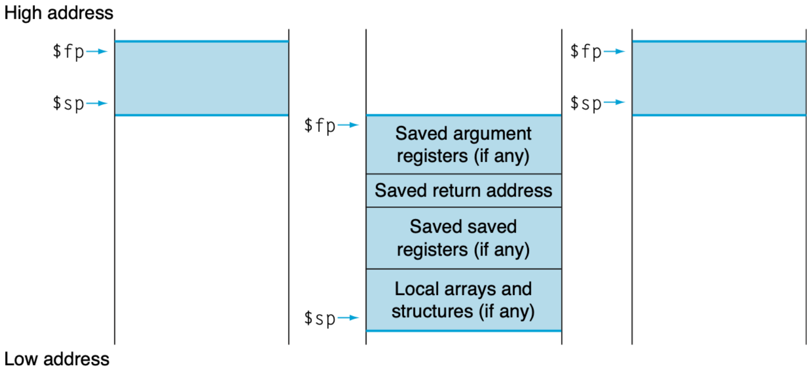
```
int fact (int n){
   if (n < 1) return (1);
   else return (n * fact(n – 1));
   }
```

```
100. fact: addi $sp, $sp, –8
104.       sw $ra, 4($sp)
108.       sw $a0, 0($sp)

112.       slti $t0, $a0, 1
116.       beq $t0, $zero, L1

120.       addi $v0, $zero, 1
124.       addi $sp, $sp, 8
128.       jr $ra
```

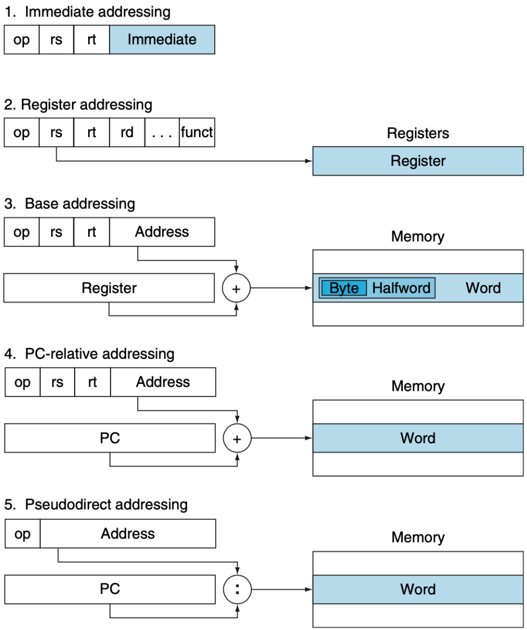
```
132. L1: addi $a0, $a0, –1
136.     jal fact

140.     lw $a0, 0($sp)
144.     lw $ra, 4($sp)
148.     addi $sp, $sp, 8

152.     mul $v0, $a0, $v0
156.     jr $ra
```

MIPS Branching Instructions

The stack is also used to store variables that are local to the procedure but do not fit in registers, such as local arrays or structures (activation record).


High address
$fp
$sp
$fp
Saved argument registers (if any)
Saved return address
Saved saved registers (if any)
Local arrays and structures (if any)
$sp
$fp
$sp
Low address

MIPS Addressing



Improving code

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

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

Improving code

Assuming that base address of array is in $a0 and size is in $a1

Array version

```
       move $t0, $zero
loop1: sll $t1, $t0, 2
       add $t2, $a0, $t1
       sw $zero, 0($t2)
       addi $t0, $t0, 1
       slt $t3, $t0, $a1
       bne $t3, $zero, loop1
```

Pointer version

```
       move $t0, $a0
loop2: sw $zero, 0($t0)
       addi $t0, $t0, 4
       sll $t1, $a1, 2
       add $t2, $a0, $t1
       slt $t3, $t0, $t2
       bne $t3, $zero, loop2
```

Improving code

Assuming that base address of array is in $a0 and size is in $a1

Array version

```
        move $t0, $zero
loop1:  sll $t1, $t0, 2
        add $t2, $a0, $t1
        sw $zero, 0($t2)
        addi $t0, $t0, 1
        slt $t3, $t0, $a1
        bne $t3, $zero, loop1
```

Pointer version

```
        move $t0, $a0
loop2:  sw $zero, 0($t0)
        addi $t0, $t0, 4
        sll $t1, $a1, 2
        add $t2, $a0, $t1
        slt $t3, $t0, $t2
        bne $t3, $zero, loop2
```

Instructions for calculating address of the last element of the array need not to be inside the loop

Improving code

Assuming that base address of array is in $a0 and size is in $a1

Array version

```
        move $t0, $zero
loop1:  sll $t1, $t0, 2
        add $t2, $a0, $t1
        sw $zero, 0($t2)
        addi $t0, $t0, 1
        slt $t3, $t0, $a1
        bne $t3, $zero, loop1
```

Pointer version

```
        move $t0, $a0
        sll $t1, $a1, 2
        add $t2, $a0, $t1
loop2:  sw $zero, 0($t0)
        addi $t0, $t0, 4
        slt $t3, $t0, $t2
        bne $t3, $zero, loop2
```

MIPS Multiplication and Division

```
mult $s0, $s1 # Multiply the numbers stored in these registers.
              # This yields a 64 bit number, which is stored in two
              # 32 bits parts: “Hi” and “Lo”

mfhi $t0 # loads the upper 32 bits from the product register

mflo $t1 # loads the lower 32 bits from the product register
```

MIPS Multiplication and Division

div $s0, $s1 # Hi contains the remainder, Lo contains quotient

mfhi $t0 # remainder moved into $t0

mflo $t1 # quotient moved into $t1

Floating point in MIPS

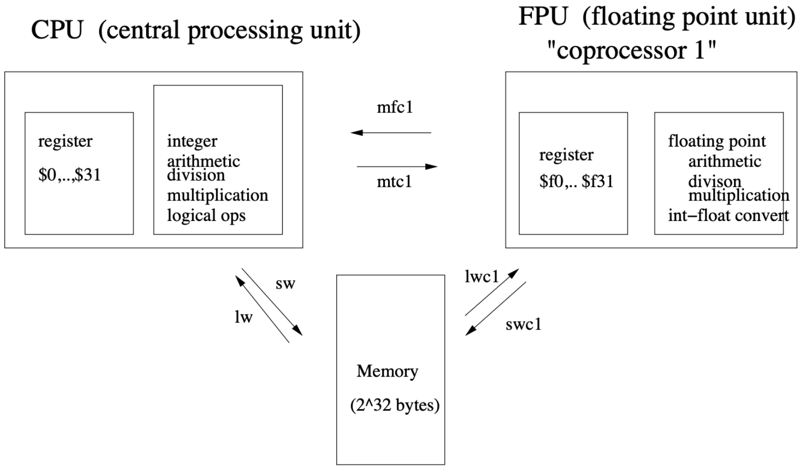
Special circuits and registers are needed for floating point operations.

The floating point operations are carried out on a physically separate chip called the floating point coprocessor or floating point unit (FPU).

The FPU for MIPS, called *coprocessor 1*, has a special set of 32 registers for floating point operations, named \$f0, \$f1, ... \$f31.

Double precision numbers require two registers. These are always consecutive registers, beginning with an even number register (\$f0, \$f2, etc).

Floating point in MIPS



Slide credit: Michael Langer

Floating point in MIPS

```
add.s $f1, $f0, $f1    # single precision add
sub.s $f0, $f0, $f2    # single precision sub
add.d $f2, $f4, $f6    # double precision add
sub.d $f2, $f4, $f6    # double precision sub
```

Following instructions are invalid

```
add.s $s0, $s0, $s1
add $f0, $f2, $f2
```

MIPS Pseudoinstructions

A common variation of assembly language instructions often treated as if it were an instruction in its own right.

move \$t0, \$t1

⇒ add \$t0, \$zero, \$t1

li \$s0, 0x3BF20

⇒

```
    lui $at, 0x0003
    ori $s0, $at, 0xBF20
```

Intel x86

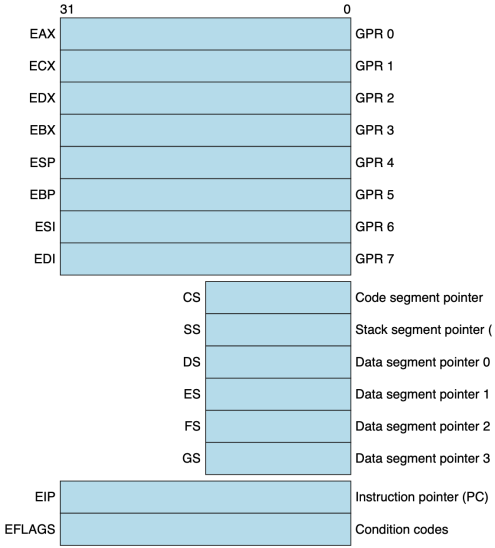
ARM and MIPS were the vision of single small groups in 1985.

x86 is the product of several independent groups who evolved the architecture over 30 years.

- **1978:** The Intel 8086 architecture was announced, 16-bit architecture, 16 bits wide register with dedicated usage.
- **1982:** The 80286 extended the 8086 architecture by increasing the address space to 24 bits.
- **1985:** The 80386 extended the 80286 architecture to 32 bits with 32-bit registers. The added instructions make the 80386 nearly a general-purpose register machine.
- **1989–95:** 80486, Pentium, and Pentium Pro.
- **1997–2001:** Pentiums were expanded with MMX (Multi Media Extensions) and SSE (Streaming SIMD Extensions).
- **2003:** From 32 bits to 64 bits transition by AMD.
- Intel adopted 64 bit version with further improvements.

Intel x86

80386 register set



Intel x86

Important aspects

- The arithmetic, logical, and data transfer instructions are two-operand instructions.
- The arithmetic and logical instructions must have one operand act as both a source and a destination.
- One of the operands can be in memory.
- Support for 8 bit, 16 bit, and 32 bit data.
- Conditional branches on the x86 are based on condition codes or flags, like ARM.
- What the x86 lacks in style, it makes up for in quantity.