Instruction Set Principles

Chapter 2 - Computer Architecture : *A Quantitative Approach* - Hennessy & Patterson

Instruction Set Architecture (ISA)

- **ISA is the systems programmer's view of a computer.**
- **Different broad classes of ISA.**
  - **General purpose and special purpose ISAs.**
  - **Complex ISA and reduced ISA.**
  - **Classification base on source and destination of operands in a CPU operation.**

Source and Destination Operands

- **Stack Architecture: zero(0) address architecture.**
- **Accumulator Architecture.**
- **Register-Memory Architecture.**
- **Load-Store Architecture.**

Stack Architecture

- **There is a small hardware stack in the processor. ALU operands are fetched from the stack and the result is pushed back in the stack.**
- **No operand address is specified with the operation (zero address).**
- **Push and pop instructions are used to transfer data between the processor stack and the memory.**
- **Instruction sizes of ALU operations is reduced.**

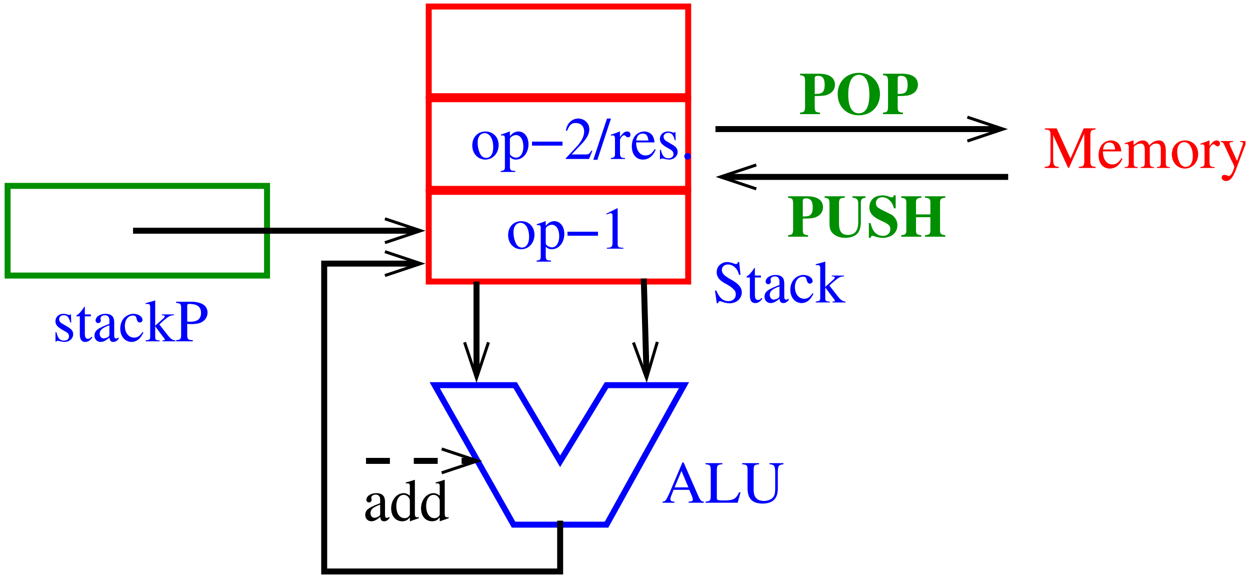

Figure 1: **Stack Architecture**

Accumulator Architecture

- **One operand and the destination of an operation is a special register called accumulator. The accumulator address is implicit.**
- **The other operand comes (often) from the memory. The address may be direct or register indirect.**

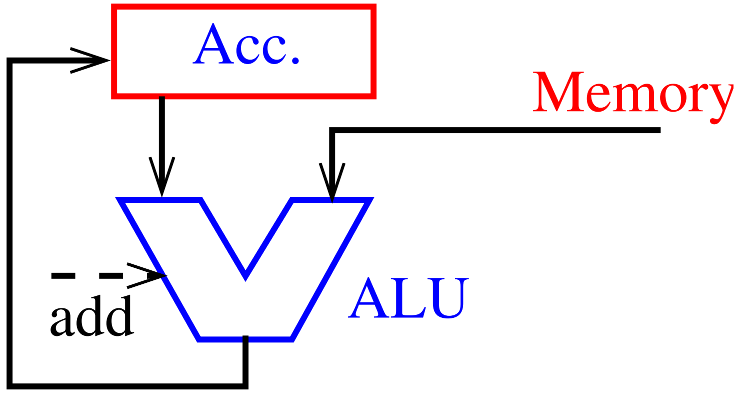

Figure 2: **Accumulator Architecture**

Register-Memory Architecture

- **One operand and the destination of an operation is a general-purpose-register (GPR). The GPR address is explicit but short.**
- **The other operand comes (often) from the memory. The address may be direct or indirect.**

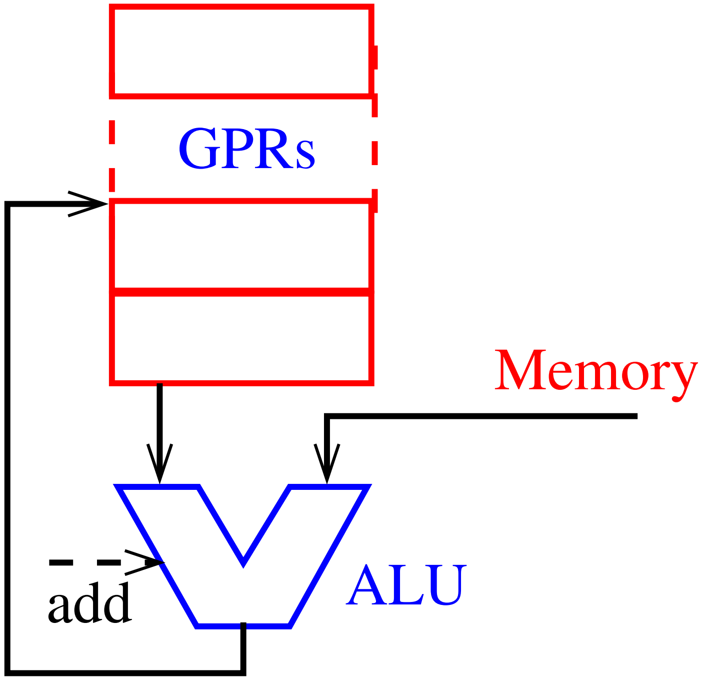

Figure 3: **Register-Memory Architecture**

Load-Store Architecture

- Both operands and the destination of an operation are general-purpose-registers (GPRs). The GPR addresses are explicit but short.
- The memory is accessed only to load a GPR or to store some value from a GPR.

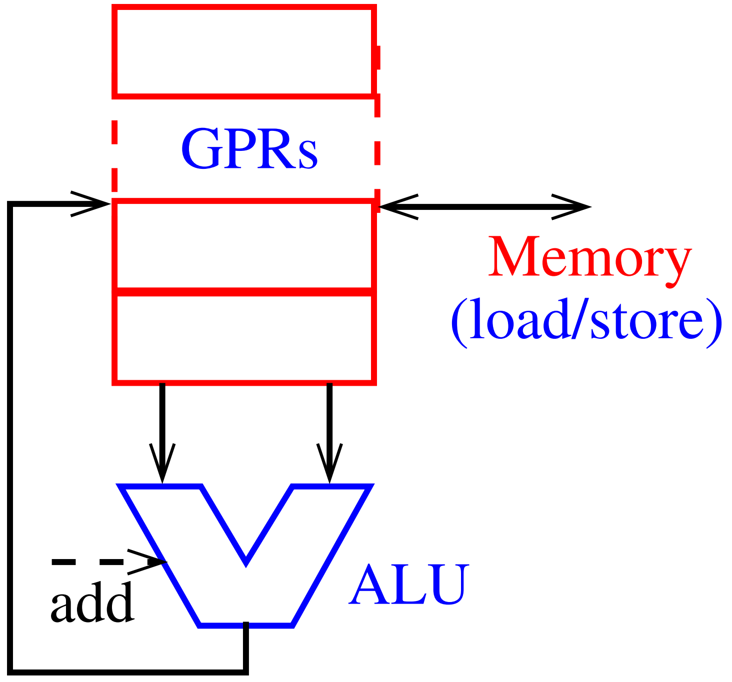

Figure 4: **Load-Store Architecture**

An Example

Consider the instruction $c = a + b$. The assembly code for diferent architectures may look as follows.

Stack	Acc.	Reg.- Mem.	Load-Store
push a	load a	load r1, a	load r1, a
push b	add b	add r1, b	load r2, b
add	store c	store r1, c	add r3, r1, r2
pop c			store r3, c

Note

- **Some CPU may have special purpose registers (original Intel86 registers were not really general purpose).**
- **Modern architectures do not support both operands from the memory.**
- **Early computers were mostly accumulator or stack architectures, but most mordern machins (except special processors) have GPRs.**

Advantages of GPRs

- **Numbers of load/stores are less than the CPU operations.**
- **GPR addresses bits are fewer in number (small number of GPRs in the CPU) that reduces the instruction size.**
- **GPRs are faster to acces (within the CPU). Less memory access.**
- **Compiler can utilize GPRs more efficiently for expression evaluation, parameter passing, return value from function etc.**

An Example

a*b - b*c - a*c

Stack	GPRs
push a	load r1, a
push b	load r2, b
mult	mult r3, r1, r2
push b	load r4, c
push c	mult r5, r2, r4
mult	sub r6, r3, r5
sub	mult r7, r1, r4

An Example

Stack	GPRs
push a	sub r8, r6, r7
push c	
mult	
sub	

Note

- **Number of memory access are more in case of stack machine.**
- **The order of evaluation is difficult to change in a stack machine (though there are independent operations).**
- **It may be necessary to change the order of instructions for better utilization of the pipeline.**

Another Order on GPR Architecture

a*b - b*c - a*c

---

load r1, a

load r2, b

load r4, c

mult r3, r1, r2 #**Depends on r2**

mult r5, r2, r4

mult r7, r1, r4

sub r6, r3, r5 #**Depends on r5**

sub r8, r6, r7 #**Depends on r6**

GPRs versus SPRs

- **A compiler can allocate variables in GPRs if there are good number of them.**
- **SPRs can be better utilized in hand-optimized code.**

Number of GPRs

- Registers for expression evaluation.
- Registers for parameter passing (often parameters are small in number).
- Registers for the value returned by a function.
- Register for return address.
- Registers for variable allocation (caller(callee) saved before (within) function call).

Operands in ALU Instructions

Op.	MemOp.	Example	Arch.
3	0	add r1, r2, r3	Alpha
3	3	add d1(r1), b, d2(r2)	VAX
2	1	add r1, d(r2)	Intelx86

Memory Addressing

- **Memory addressing for data access.**
- **Memory addressing for changing the flow of execution.**

Multibyte Data

- The memory is **byte addressable** and the address is 32-bit.
- An integer is 4B long (say): **0x04030201** is in a register -

31			0
0000 0100	0000 0011	0000 0010	0000 0001

Multibyte Data

In the memory there are two different ways to store the data from the location 0xAB $\cdots$0.

- **Little Endian (LSB):**

0xAB $\cdots$0	0xAB $\cdots$1	0xAB $\cdots$2	0xAB $\cdots$3
0000 0001	0000 0010	0000 0011	0000 0100

- **Big Endian (MSB):**

0xAB $\cdots$0	0xAB $\cdots$1	0xAB $\cdots$2	0xAB $\cdots$3
0000 0100	0000 0011	0000 0010	0000 0001

Character Data

- **Little Endian (LSB):**

0xAB ···0	0xAB ···1	0xAB ···2	0xAB ···3
A	B	C	D

- **This data in the register will be**

31			0
D	C	B	A

Register cannot be compared directly for string comparison.

Aligned and Misaligned Data

- **A data of size 2^k byte is said to be aligned if the starting address has k least significant bits all zeros. Otherwise it is misaligned.**
- **Single byte data is always aligned.**
- **A 2-byte, 4-byte and 8-byte data are aligned if the address bits $b_0 = 0$, $b_1b_0 = 00$ and $b_2b_1b_0 = 000$ respectively.**

Misaligned Data Access

- Some computers **do not allow misaligned data access**.
- **Misaligned data access** is **slower** even if it is permitted.
- In some machine **misaligned data access assembly instruction** is translated to **more than one machine instruction**.

Misaligned Data Access: an example

An access to 4-bytes of data from the address 0xE (i.e. 0xE $\cdots$ 0x11) amounts to

- Two 4-byte data fetch, 0xC $\cdots$ 0xF and 0x10 $\cdots$ 0x13.
- The extraction of the required bytes.

0xC	0xD	0xE	0xF	0x10	0x11	0x12	0x13
		0th	1st	2nd	3rd		

Misaligned Data Access: an example

The misaligned instruction mload r1, 0xE is translated to

```
load r2, 0xC
load r1, 0x10
andi r1, 0xFFFF0000
andi r2, 0xFFFF
and r1, r1, r2
```

Specifying the Data Address

Addr. Mode.	Example	Meaning
Register_3	add r1, r2, r3	r1 = r2 + r3
Register_2	add r1, r2	r1 = r1 + r2
Immediate	add r1, $4	r1 = r1 + 4
Displacement	add r1, 100(r2)	r1 = r1 + M[r2+100]

Displacement is also called by some author as indexed[a].

[a] Computer Organization by Hamacher, Vranesic and Zaky, 3rd Ed.

Specifying the Data Address

Addr. Mode.	Example	Meaning
Reg.Indirect	add r1, (r2)	r1 = r1 + M[r2]
Indexed	add r1, (r2+r3)	r1 = r1 + M[r2+r3]
Direct/ Absl.	add r1, (1000)	r1 = r1 + M[1000]
Memory Indirect	add r1, ((r2))	r1 = r1 + M[M[r2]]

Specifying the Data Address

Addr. Mode.	Example	Meaning
AutoInc.	add r1, (r2)+	r1 = r1 + M[r2] r2 = r2 + 4
AutoDec.	add r1, -(r2)	r2 = r2 - 4 r1 = r1 + M[r2]
Scaled	add r1, 100(r2, r3, 4)	r1 = r1 + M[r2 + 4*r3 + 100]

Effect of Addressing Mode

- Complex addressing mode may increase the average CPI and add to the complexity of hardware implementation.
- Addressing modes with side-effect complicates the restarting of an instruction after page-fault.
- There is wide variation in instruction length due to complex addressing modes.
- Some addressing modes may have poor utilization by a compiler.

Measurement with Addressing Mode

Measurement on VAX machine. Most frequently used addressing modes.

Displacement

Immediate

Register Indirect

Scaled

Sizes of Displacement and Immediate Data

- These two sizes affect the instruction length.
- The measurement shows that there are large number of small and also a good number of larger displacements - these are due to different storage areas for data.
- There are three essential use of immediate data - constant in an expression, comparison for branch and initilization of variables (register).
- Often the immediate constants are small (unless it is for an address).

Special Purpose Addressing Modes

- **Addressing mode for circular buffer in DSP - start-address and end-address registers with every address register.**
- **Bit reverse addressing for FFT calculation.**

Data Addressing: Summary

- **Displacement, immediate and register indirect addressing modes are most important.**
- **Size of displacement should be 12-16 bits and the size of immediate data should be 8-16 bits.**
- **Other than postincrement/postdecrement of registers (after using the content as address), similar instructions are rarely used.**

Primitive Data Types

- **Character: 8-bit ASCII or 16-bit Unicode.**
- **Integer: 32-bit 2's complement, short (16-bit) and long (64-bit).**
- **Floating-point: single (32-bit) or double (64-bit) word IEEE standard 754.**
- **BCD: binary-coded decimal (BCD) for exact decimal arithmetic.**

Application Specific Data

- **vertex** (x, y, z and w, each 32-bit floating-point), **pixel** (32-bit, four 8-bit channel, R, G, B and A).
- **fixed-point** - binary point between the **sign** and the **msb** - fraction within the range -1 to $+1$.

Basic Operation Types

- **Arithmetic and Logical**: basic integer and logical operations.
- **Data Transfer**: memory-register, register-register.
- **Control**: branch, jump, call, return
- **IO**: IO instructions for separate space processor.
- **Systems**: systems call, memory management.
- **Floating-point**: floating point operations.

Application Specific Operations

- **String**: operations on strings e.g. string copy, move, compare, search etc.
- **Graphics**: vertex, pixel operations.
- **Decimal**: operations on BCD data and conversion to other format.

Control Flow Instructions

- **Conditional branches.**
- **Unconditional jumps.**
- **Procedure calls.**
- **Return from procedure.**
- **System calls or software interrupts or traps.**

Control Flow: Addressing Modes

Addr. Mode.	Example	Meaning
Implicit		PC = PC + k
Direct Address	jmp 1000	PC = 1000
Register Indirect	jmp r1	PC = r1
Memory Indirect	jmp (r1)	PC = M[r1]
PC Relative	jmp (-100)	PC = PC - 100

Jump target not Known at Compile Time

The actual jump address is loaded from the memory to the specified register at run-time.

- Case and switch statement in a programming language.
- Virtual function call in object-oriented languages.
- Calling a function through function pointer (passes as parameter) or higher-order and polymorphic function in functional languages.
- Dynamically shared libreries.

Measurements: PC-relative Addressing

- **75% branches are in the forward direction.**
- **Most displacements are less than 8-bits.**

Branch Conditions

- Often the branch conditions are simple and large number of them are comparison with zero.
- The most frequent comparisons are '$\leq$', '$<$' and '$=$'.
- They ae treated as special cases in some architecture.
- DSPs has special instruction to repeat a set of instructions.

Specifying Branch Conditions

- Condition bits in a special register - condition is set free of cost, proble with out of order execution.
- Result of a test saved in a register which is tested for condition - uses a register.
- Compare and branch instruction - execution of only one instruction, the instruction may be complex for pipeline implementation.

Procedure Call and Return

- The state of the caller is to be saved - return address, status word, and may be the CPU registers.
- The return address may be saved on the stack or in a link register or in a GPR.
- Some old architecture used to save and restore (on return) all CPU registers. In modern architecture compiler generates code for store and load.

Who Saves the Registers?

- **Caller saving** those registers that it wants to preserve across the call.
- **Callee saving** a register before it wants to use.
- **Global variable** in a register (load/store).
- **Convension** specified by **application binary interface (ABI)**.

Encoding Instructions

- **Specification of operation: opcode.**
- **Number of memory and register operands.**
- **Addressing modes for each operands: address specifier.**

Different Encodings

- **Variable length instructions: VAX, Intel 80x86 etc**
- **Fixed Length: MIPS, PowerPC etc.**
- **Hybride: IBM 360/70, MIPS16 etc.**

Variable Length Instruction: An Intel Example

pushl %ebp	**0x55**	**Operation + Reg. Address**

0	**1**	**0**	**1**	**0**	**1**	**0**	**1**
push r32				**ebp**			

000	**001**	**010**	**011**	**100**	**101**	**110**	**111**
eax	**ecx**	**edx**	**ebx**	**esp**	**ebp**	**esi**	**edi**

Variable Length Instruction: An Intel Example

pushl -512(%ebp,%ebx,4)

0xFF B4 9D 00 FE FF FF

FF	B4			9D			00 FE FF FF
FF	10	110	100	10	011	101	-512

Opcode (FF 110), 32-bit disp (10), base + index (100), scale 4 (10), index - ebx (011), base - ebp (101)

Variable Length Instruction

Length of an Intel 80x86 instruction may vary between 1 - 17 bytes.

Fixed Length Instruction: MIPS Example

ALU Instruction (R-format)

op(6)	rs(5)	rt(5)	rd(5)	shmat(5)	funct(6)

Fixed Length Instruction: MIPS Example

Branch and Immediate Data Instructions

(I-format)

op(6)	**rs(5)**	**rt(5)**	**imm./displ. (16)**

Fixed Length Instruction: MIPS Example

Jump Instructions (J-format)

op(6)	**Jump address (26)**

RISC in Embedded Application

- **Smaller and more compact code - restriction on memory size.**
- **Two instruction formats.**
- **Lesser number of operations, small address and immediate data size, fewer registers etc.**

RISC in Embedded Application

- **IBM keeps the same instruction set but compresses it in the memory.**
- **Hardware decompresses it at the ime of fetch and puts 32-bit instruction in the cache.**
- **No new compiler is required but the effective cache utilization is less.**
- **How to fetch the next instruction on jump/branch?**

Review

- **General purpose register (GPRs) with load/store architecture.**
- **Good support of displacement (12-16 bit), immediate (8-16 bit) and register indirect addressing modes.**
- **Support for 8-, 16-, 32, and 64-bit integers and 64-bit IEEE 754 floating-point numbers.**
- **Support for simple instructions e.g. load, store, add, subtract, move, shift etc.**

Review

- **Compare less, equal, not equal, PC-relative banch (8-16 bit offset), jump, call, return.**
- **Fixed instruction encoding for performance or variable encoding for small code size.**
- **Addressing modes should be orthogonal.**
- **At least 16 or 32 GPRs.**

MIPS64: Registers

- **GPRs**: $R0, R1, \cdots, R31$ **- each of size 64-bits.**
- **FPRs**: $F0, F1, \cdots, F31$ **- IEEE 754 single and double precision formats.**
- **The value of $R0$ is always zero (0).**
- **A few special registers.**

MIPS64: Data Type

- **GPRs**: $R0, R1, \cdots, R31$ **- each of size 64-bits.**
- **FPRs**: $F0, F1, \cdots, F31$ **- IEEE 754 single and double precision formats.**
- **The value of** $R0$ **is always zero (0).**
- **A few special registers.**

MIPS64: Data Type

- **Integer: 8-, 16-, 32-, 1nd 64-bit.**
- **Floating-point: 32- ans 64-bit IEEE 754.**

MIPS64: Addressing Modes

- **Immediate: 16-bit.**
- **Displacement: 16-bit.**
- **Register Indirect: free of cost.**

MIPS64: Instruction Formats

- **R-format: register ALU operations.**
- **I-format: Immediate, branch etc.**
- **J-format: jump.**

MIPS64: Examples

LB	**Load byte**
LH	**Load half-word**
LW	**Load word**
LD	**Load f-word**
LWU	**Load word unsigned**

MIPS64: Examples

L.S Load floating-point single precision

L.D Load floating-point double precision

MIPS64: Examples

L (**load**), **S** (**store**), **ADD** (**add**), **DADD** (**double add**), **DADDU** (**double add unsigned**), **S** (**store**), **LUI** (**load upper immediate**), **DSLL** (**shift left logical**), **DSLT** (**set less than**),

MIPS64: Control Flow Instructions

- **J target**: PC[27 .. 0] = 4*target - the jump "target" is 26-bits, an instruction is aligned in word boundary, the "target" is multiplied by 4 and loaded in PC. Higher bits of PS unchanged(?).
- **JAL name**: R31 = PC+4; PC[27 .. 0] = 4*target - the return address is saved in the link register.
- **JALR R1**: R31 = PC+4; PC = R1 - register indirect jump and link.

MIPS64: Control Flow Instructions

- **JR R1: PC = R1 - register indirect jump.**
- **BEQZ R1, name: if (R1 == 0); PC = PC+4 + 4*name - branch eq. zero.**
- **MOVZ R1, R2 R3: if (R3 == 0); R1 = R2 - conditional move.**