Instruction Set Architecture

Introduction:

Instruction Set:
Interface between software and hardware visible to programmer

Steps to design Instruction Set for advanced computer:

- what is the available alternatives
- need to asses each alternative
- need quantitative method
- how does the compiler affect ISA

Measurements for evaluating ISA alternatives:

- depend on program and compiler
- use benchmarks (SPEC)
- Using dynamic measurements number of times each events occurs

N. Mekhiel
**Instruction Set Classification** Classification based on Internal storage of operands in CPU:

- **Stack:**
  Operands accessed from the top of stack
  0 Address
  Example: push A; Mem(SP)=A, SP+1

- **Accumulator**
  One operand is implicitly the Accumulator
  1 Address
  Example: ADD B; A=Acc + Mem(B)

- **set of Registers**
  Operands are in registers
  2 Address
  Example: ADD R1, R2 ; R1= R1 + R2
  3 Address
  ADD R1, R2, R3 ; R1= R2 + R3

N. Mekhiel
Example to Compare Different ISA

C=A+B;

- **Stack**
  
  PUSH A;
PUSH B;
ADD
POP C
Disadvantages: Top of the stack is a bottleneck

- **Accumulator**
  
  LDA A; Acc=A
ADD B ; Acc=A+B
STA C ; C= A+B
Disadvantage: Memory traffic

- **General Purpose Registers ”GPR”**
  
  LW R1, A ; R1=A
LW R2, B ; R2=B
ADD R3, R1, R2 ; R3=A+B
SW C, R3 ; C=A+B
Disadvantage: cost of registers

N. Mekhieh
Advantages of Using GPR

- Registers are faster than memory
- Easier for Compiler to Use
  Example: A.B - C.D - E.F Could be evaluated in any order (reduce pipeline hazards) compared to stack that must execute instructions in order
- Registers hold variables and reduce memory traffic, reduce code density

Types of GPR Architectures:

- ALU operations with TWO OPERANDS
  ADD R1, R2
  ALU operations with THREE OPERANDS
  ADD R1, R2, R3

N. Mekhiel
• Memory Addresses in ALU instructions:
  – 0 memory operands (LOAD/STORE)
    ADD R1, R2, R3
    Advantages: Simple, fixed length instructions are faster to encode, simple code generation, instructions have same number of clock cycles to execute
    Disadvantages: Higher instruction count, short instructions waste bit encoding
  – Register-Memory (1 memory address, 2 register operands)
    ADD R1, B ; R1 = R1 + M(B)
    Advantages: no need to load data first, good code density, easy to encode instructions
    Disadvantages: could destroy source operand, long instruction encoding causes limited number of registers, variable number of clocks to execute instruction (depends on operand location)

N. Mekhiel
Memory Addressing
How memory addresses are interpreted: means what objects are accessed (byte, half word, word, double word)
Byte Ordering:
How do byte addresses map onto words? can a word be placed on any byte boundary?

• Big Endian:
  Words address = address of most significant byte (MIPS, IBM, Motorola)

• Little Endian:
  Words address = address of least significant byte (Intel, DEC)

<table>
<thead>
<tr>
<th></th>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Little Endian</td>
<td>3 2 1 0</td>
<td></td>
</tr>
<tr>
<td>Big Endian</td>
<td>0 1 2 3</td>
<td></td>
</tr>
</tbody>
</table>

N. Mekhiel
Object Alignment:

Objects larger than byte must be aligned. Objects of size S bytes at byte address A is aligned if:
A mod S = 0
Example: Accessing a word (4 bytes) at address 13
13 mod 4 = 1 not aligned
Example 2: half word at address 12:
12 mod 2 = 0 is aligned
If memory access is not aligned, it will need 2 references (slower).
Need alignment network to support byte/half word access in registers
Example: loadH R1, 40(R3)

— Bit31.......Bit0
R1= A—B—X—X ; X =no change

N. Mekhiel
**Addressing Modes** How instructions specify address of object to access
Effective address: actual memory address used to access memory

**Types of addressing Modes:**

- **Register**
  To access variables;
  Example: ADD R4, R3 ; R3 = R4 + R3

- **Immediate**
  To access Constants;
  Example: ADD R4, #3 ; R4 = R4 + 3

- **Displacement**
  To access local variables;
  Example: ADD R4, 200(R1) ; R4 = R4 + M(200+R1)

- **Register Indirect**
  To implement pointers;
  Example: ADD R4, (R1) ; R4 = R4 + M(R1)

N. Mekhiel
Addressing modes:

- **Indexed**
  To access arrays;
  Example: \( \text{ADD R4, (R1+R2)} ; \ R4 = R4 + M(R1+R2) \)  
  R1 base, R2 index

- **Direct**
  To access static data;
  Example: \( \text{ADD R4, (1001)} ; \ R4 = R4 + M(1000) \)

- **Memory Indirect**
  To access \( p \);
  Example: \( \text{ADD R4, (R3)} ; \ R4 = R4 + M(M(R3)) \)

- **Autoincrement**
  To step through arrays;
  Example: \( \text{ADD R4, (R1)+} ; \ R4 = R4 + M(R1) \) and \( R1 = R1 + d \) (size of element)

N. Mekhiel
Addressing modes:

- **Autodecrement**
  To implement stack;
  Example: `ADD R4, -(R1); R1 = R1 - d; R4 = R4 + M(R1)`

- **Scaled**
  To index arrays;
  Example: `ADD R4, 100(R1)[R3]; R4 = R4 + M(100 + R1 + R3d)`

Addressing modes significantly reduce Instruction counts, but may also increase CPI for machine that implement them.

N. Mekhiel
Need to make design decision about what addressing modes should ISA supports?
We should make our decision based on Quantitative measurements of % usage of each addressing mode (use benchmarks).
Results of SPEC89 for VAX using tex, spice, gcc applications are:-

- Displacement = 42% on average (from 32% to 55%)
- Immediate = 33% on average (from 17% to 43%)
- Indirect = 13% on average (from 3% to 24%)
- Scaled = 7% on average (from 0% to 16%)
- Memory Indirect = 3% on average (from 1% to 6%)
- Misc = 2% on average

We should support: Displacement, immediate and Indirect which covers upto 99% of all addressing modes.

N. Mekhiel
Need to make design decision about what should be the size of displacement address measurements of % displacement size (use benchmarks).

Results of SPEC2000 for VAX using tex, spice, gcc applications are:-

- Displacement size of 16 bits captures most of used displacements (99%)

- Note: use cumulative distribution for measuring % usage:

<table>
<thead>
<tr>
<th>Disp</th>
<th>%use</th>
<th>%Cumulative</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>26</td>
<td>26</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>28</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>36</td>
</tr>
</tbody>
</table>

Design decision taken: use 16 bit displacement

N. Mekhiel
Need to make design decision about what should be the size of Immediate?
measurements of immediate size in arithmetic operations (comparisons, moves to registers, ALU Ops).

<table>
<thead>
<tr>
<th></th>
<th>loads</th>
<th>compare</th>
<th>ALU op</th>
</tr>
</thead>
<tbody>
<tr>
<td>frequency</td>
<td>10%</td>
<td>87%</td>
<td>70%</td>
</tr>
</tbody>
</table>

Design Decision: must support ALU OP, Compare using immediate addressing mode.

What is the value of size of Immediate
from measurements of gcc, tex, spice are:-
50% to 60% fit within 8 bits
75% to 80% fit within 16 bits

Design Decision: Use 8 or 16 bits for size of Immediate
N. Mekhiel
Operations in ISA

Typical operations: Data transfer, arithmetic&logic operations, control flow
Simple instructions are the most used
Example: 80X86 Instructions

- Load = 20 %
  - conditional branch = 20 %
  - compare = 16 %
  - store = 12 %
  - add = 8 %
  - sub = 5 %
  - move register to register = 4 %
  - call = 1 %
  - return = 1 %

Design Decision: support simple instructions to reduce CPI and complexity (RISC)

N. Mekhiel
What types of control flow should ISA supports? Types and frequency of control flow using SPEC92:-
call\&return = 13 \% for INT, 11 \% FP
JUMP = 6 \% INT, 4
Cond branch = 81 \% INT, 87 \% FP
In jump or branch using PC relative to optimize the number of bits needed for branch displace-
ment. Destination must be close to current in-
struction.

**What should be the value of branch displace-
ment in bits?**
Most branch displacement Less Than 10 bits (4-8 bits)

N. Mekhiel
How to specify branch condition?

- **Condition code:**
  ALU operations set bits of condition code
  Advantages: CC sets for free
  Disadvantages: -Instructions could set CC hap-hazardly.
  -uses an extra state.
  Example:
  ```
  ADD R1, R2, R3
  bz label
  ```

- **Condition register:**
  Test arbitrary register for result of a comparison
  Advantages: simple, predictable
  Disadvantages: uses extra register
  Example:
  ```
  slt R1, R2, R3
  beq R1, label
  ```

N. Mekhiel
How to specify branch condition?

- **Compare and branch:**
  
  Compare is part of branch.
  
  Advantages: one instruction
  
  Disadvantages: too much for Inst (affect CPI)
  
  Example: `bgt R1, R2, label`

**Types of Compares to be supported:**

Less than/greater than (7% INT, 40% FP)
Greater than/less than or equal (7% INT, 23% FP)
Equal/not equal (INT 86%, 37% FP)

Most important and used type of compare is equal/not equal (compare with 0)

N. Mekhiel
Handling branch destination in call/return
Cannot use PC relative because destination address is not known at run time.
Use a register that has target address, and use register indirect jumps:
jump (R2)

Methods of saving registers in procedure calls and returns

- caller saving:
  more conservative. P1 uses X and calls P2, then P2 calls P3 that uses X.
  Value of X is saved for P1 (if callee saving P2 does not save X).

- callee save:
  more optimal for some applications.

N. Mekhiel
Review

- Use GPR with load/store
- Support the following addressing modes:
  Displacement with address offset size=16 bits;
  Immediate with size 8 to 16 bits; Register indirect.
- Support simple instructions because they dominate the number of executed instructions as:
  load; store; add; subtract; and; shift; compare equal; compare not equal; branch with PC relative of size 8 bits; jump; call and return.
- Use condition register for testing conditions
- Make instruction length fixed (faster decoding)

N. Mekhiel
Types and size of Operands

Types and sizes are:

- **Integer:**
  8 bits = byte
  16 bits = half word
  32 bits = word

- single precision floating point:
  32 bits

- double precision floating point:
  64 bits using IEEE 754 Standard

- character:
  ASCII = 7 bits

- decimal:
  packed decimal or BCD = 4 bits

N. Mekhiel
Design Decision: what types of data should be supported by ISA?
We have the following results from benchmarks:
Double word: INT = 0 %, FP = 69 %
Word: INT = 74 %, FP = 31 %
Halfword: INT = 19 %, FP = 0 %
Byte: INT = 7 %, FP = 0 %

Design Decision: For INT Support 8 bit, 16 bit, 32 bit operands
For FP Support 32 bit, 64 bit operands.

bf Instruction Format Affect Size of Program and Processor Implementation
Question: How to encode the addressing modes with operations?
Need the following:

- as many registers and addressing modes as possible
- small instruction size
- instruction size must be fixed (multiple bytes)

N. Mekhiel
Instruction Formats:

- **Variable:**
  best for supporting many addressing modes
  Example: in VAX ADD R3, 737(R2), (R1)

- **Fixed:**
  Opcode has operation and addressing mode.
  Easy to decode (faster) but size of Instruction is not optimum. Suitable for RISC
  Example: MIPS, SPARC, PowerPC: ADD R1, R2, R3

- **Hybrid:**
  Provide multiple fixed size Instructions.
  Example: Intel 8086

N. Mekhiel
The Role of Compilers
Compiler translates HLL to Assembly (Uses ISA Available).

\[ \text{Performance(ExecutionTime)} = \text{NumberOfInstructions} \times \text{NumberOfClockCyclesPerInstruction} \]

Compiler selects types of Instructions, number of Instructions for each task.

Optimization Methods

- **Use Registers**
  If two instances of expression compute same value, use a register to evaluate the value once (eliminate second computation).

- **Register Allocation:**
  Use algorithm to find best set of registers to allocate to variables (graph coloring).
  Graph coloring needs at least 16 registers.

- **Code Sequence:**
  Use code sequence to reduce pipeline hazards (Out of Order execution).

N. Mekhiel
• Dealing with variables referenced by Pointers
  Compiler should not assign variable to register if a pointer refer to it.
  Aliasing problem: If A≡5 and if we use a register for a, R1≡A≡5 while a pointer is used to reference A, then
  P≡7 will cause A to have different value. (Heap allocated objects cannot use registers).

  N. Mekhiel
The ISA and Ease of Compilers
ISA should make compilers easier using the following Rules:-

• **Regularity:**
  Orthogonal: make three aspects of ISA orthogonal (Independent).
  for each operation, all addressing modes could be used on all types of operands.
  It makes code generation easier, no special registers, few special cases.

• **No Special Features:**
  No special features to match a language (works only for few cases one language).

• **Predictable cost:**
  Using registers versus memory, cache and pipelining makes it very difficult for compiler to decide what sequence to use.

• **Dealing with Constants**
  Quantities known at compile time should be treated as constants (no change).

N. Mekhiel
MIPS Architecture

- Use GPR with load-store

- Support these addressing modes: displacement (with address offset=12-16 bits); immediate (size 8-16 bits); and register indirect
  lw R1, 30(R2); R1 = M(R2+30)
  if R2=R0, we have absolute addressing mode. If 30 is replaced by 0, we have register deferred

- support the following data sizes: 8 bits, 16 bits, 32 bits, and 64 bits for double precision using IEEE 754 FP.

- support the following instructions: load, store, add, subtract, move register and shift

- uses compare equal, compare not equal, compare less with PC relative (8 bits long), call and return

- use fixed instruction length for faster decoding

- use at least 16 registers and orthogonal ISA.

N. Mekhieil
MIPS Instructions

Instruction Format

• I type
  Example: LD R1, 30(R2); R1 = R1 + M(R2+30)
  Format: Opcode=6 bits; rs= 5 bits; rt = 5bits; immediate= 16 bits
  opcode=LD ; rs=R2; RT =R1, immediate= 30

• Register type
  Format: opcode=6 bits; rs1=5bits; rt=5 bits;rd=5 bits; operation type=11 bits
  Example: ADD R1, R2, R3 ; R1= R2 + R3
  opcode = ADD; rs = R2; rt= R3; rd=R1

• J Type
  Example: JR R3; PC = (R3)
  Format: opcode=6 bits; Offset added to PC = 26 bits
  opcode JR ;

N. Mekhiel
Effectiveness of MIPS ISA

Instruction set frequency for SPEC 92 is:
load=26%, store=9%, add=14%, compare=13%,
load imm=3%, cond branc= 16%, uncond branch
call return jump= 3%, shift =4%, and = 3%,
or=5%, xor=1%.

Performance = Instruction count × CPI × clockcycle time

compare VAX performance to MIPS for SPEC
VAX has smaller instruction count, but MIPS has
faster CPI

- For example running spice:

\[\text{No - of - Instructions} - \text{MIPS} \div \text{No - of - Instruction} - \text{VAX} = 1.8\]

\[\text{CPI - MIPS} \div \text{CPI - VAX} = .25\]

Performance of MIPS is 2.5 times of VAX

- Example for FP:

\[\text{No - of - Instructions} - \text{MIPS} \div \text{No - of - Instruction} - \text{VAX} = 2.7\]

\[\text{CPI - MIPS} \div \text{CPI - VAX} = .1.\]

Performance of MIPS is 4 times of VAX

N. Mekhiel
Problems

2-4 Find the required memory bandwidth (code + data) for the following architectures:-

- Accumulator (all operations between Acc, Mem)
- Memory-Memory (all operands are in memory)
- Stack only push and pop access memory.
- Load/store (all operations occur in registers).

Assume 16 registers (4 bits to address a register)

Assume the following:-

- Opcode = 1 byte
- memory address = 2 bytes
- operands = 4 bytes
- instructions are multiple number of bytes.

Use the following application in C:-

A = B + C;

All variables are initially in memory.
2-4

1- Stack

<table>
<thead>
<tr>
<th>INSTRUCTION</th>
<th>DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH B = (1+2) B,</td>
<td>4 B</td>
</tr>
<tr>
<td>PUSH C = (1+2) B,</td>
<td>4 B</td>
</tr>
<tr>
<td>ADD = 1 B,</td>
<td>0</td>
</tr>
<tr>
<td>POP A = (1+2) B,</td>
<td>4 B</td>
</tr>
</tbody>
</table>

Total= 10 B + 12 B = 22 B BW = 22B/4 = 5.5B/CYCLE

2- ACC

<table>
<thead>
<tr>
<th>INSTRUCTION</th>
<th>DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDA B = (1+2) B,</td>
<td>4 B</td>
</tr>
<tr>
<td>ADDA C = (1+2) B,</td>
<td>4 B</td>
</tr>
<tr>
<td>STA A = (1+2) B,</td>
<td>4 B</td>
</tr>
</tbody>
</table>

Total= 9 B + 12 B = 21 B BW = 21B/3 = 7B/CYCLE
2-4
3-Load/store

<table>
<thead>
<tr>
<th>INSTRUCTION</th>
<th>DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW R1, B = (1+ .5+2) B,</td>
<td>4 B</td>
</tr>
<tr>
<td>LW R2, C = (1+ .5+2) B,</td>
<td>4 B</td>
</tr>
<tr>
<td>ADD R3,R1,R2 = (1 +3X.5)B,</td>
<td>0</td>
</tr>
<tr>
<td>SW A, R3 = (1+.5+2) B,</td>
<td>4 B</td>
</tr>
</tbody>
</table>


4-Memory

<table>
<thead>
<tr>
<th>INSTRUCTION</th>
<th>DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD A,B,C B = (1+2 +2+2) B,</td>
<td>3X 4 B</td>
</tr>
</tbody>
</table>

Total= 7 B + 12 B = 19 B  BW = 19B/1 = 19B/CYCLE
2-5 If offset length for branches and memory references are: 0, 8, 16 bits. Instruction length = 16 bits + length of offset. Using the data of frequency of data references and branches versus offset in number of bits, and instruction frequencies of benchmarks, find the average instruction length of following:

a-If permitted offsets are 0, 8, 16 including the sign bit.

b-If using fixed instruction length of 24 bits. If an offset is longer than 8 bits, another instruction is required.
results of benchmarks (SPEC89, SPEC20000 are:

<table>
<thead>
<tr>
<th>offset bits</th>
<th>data references</th>
<th>branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>17% (30.7% SPEC2000)</td>
<td>0%</td>
</tr>
<tr>
<td>7</td>
<td>57% (60.9% SPEC2000)</td>
<td>93% (85%</td>
</tr>
<tr>
<td>15</td>
<td>100%</td>
<td>100%</td>
</tr>
</tbody>
</table>

---

Instruction usage

- load: 26%
- store: 9% (10% spec2000)
- cond branch: 16% (12% spec2000)
- uncond branch: 1%
- others: 100-35-17=48%

Instruction length = 16 bits + offset

- a= 48%x(16+0) + 35%x[(17%x(16+0)
  +40%x(16+8) + 43%x(16+16)]
  +17%x[(93%x(16+8)+7%x(16+16)]=21 bits
- b= 48%x24 +35%x[17%x24+ 40%x24 +43%x48]
  + 17%x[93%x24 + 7%x48]= 27 bits
2-11 Find the effective CPI for MIPS using frequencies of benchmarks for gcc and espresso assuming:

cost of ALU operations = 1 cycle
-cost of load/store = 1.4 cycles
-cost of cond. branch (taken) = 2.0 cycles
-cost of cond branch (not taken) = 1.5 cycles
-cost of unconditional branch = 1.2 cycles

Assume that 60% of cond branches are taken from benchmarks:
load = 26%, store=9%
cond branch=16%, unconditional branch=1%
others = 100% - 52% = 48%

CPI = 48\%X1 + 35\%X1.4 
+ 16\%(60\%X2 + 40\%X1.5) 
+ 1\%X1.2 =
2-12-b Consider adding a new addressing mode to MIPS. Compiler will replace the following:

ADD R1, R1, R2

LW R4, 0(R1) or SW—

with one load/store instruction with new addressing mode.

Use the instruction frequencies of benchmarks, find performance effect of new mode if it lengthens the clock cycle by 5% and it could be used in 10% of load/store.

2-12-b- load/store frequency from benchmarks=35% (may change with applications).

Time of executing N instructions on original machine = NXCPIXClock Cycle Time
Machine using new addressing mode will eliminate 10% of 35% instructions.

Time={N-.35X.1XN}XCPIX1.05 Clock Cycle Time
=1.013 NXCPIXClock Cycle Time (slower)