Instruction Set Design
Instruction Set Architecture: to what purpose?

- ISA provides the level of abstraction between the software and the hardware
  - One of the most important abstraction in CS
  - It’s narrow, well-defined, and mostly static
  - (compare writing a windows emulator [almost impossible] to writing an ISA emulator [a few thousand lines of code])
What do we want in an ISA?

- Compact
- Simple
- Scalable (64bit)
- Spare opcodes
- Amenable to hw implementation.
- express parallelism
- Turing Complete
- Make the common case fast.
- Easy to verify
- Cost effective
- easy to compile for
- Consistent/regular/orthogonal
- Regular instruction format.
- Good OS support
  - protection
  - VM
  - Interrupts
- Easy for the programmers
Crafting an ISA

• Designing an ISA is both an art and a science

• Some things we want out of our ISA
  – completeness
  – orthogonality
  – regularity and simplicity
  – compactness
  – ease of programming
  – ease of implementation

• ISA design involves dealing in a tight resource
  – instruction bits!

• “This will go down on your permanent record”
  – ISAs live forever (almost)
  – Be careful what you put in there
Basic Questions

- Operations
  - how many?
  - what kinds?

- Operands
  - how many?
  - location
  - types
  - how to specify?

- Instruction format
  - how does the computer know what means?
  - size
  - how many formats?

\[
y = x + b
\]

destination operand
operation
source operands

0001 0100 1101 1111
Operand Location

- Can classify machines into 3 types:
  - Accumulator
  - Stack
  - Registers

- Two types of register machines
  - register-memory
    - most operands can be registers or memory
  - load-store
    - most operations (e.g., arithmetic) are only between registers
    - explicit load and store instructions to move data between registers and memory
How Many Operands?

**Accumulator:**

1 address add A acc ← acc + mem[A]

**Stack:**

0 address add tos = tos + next

**Register-Memory:**

2 address add Ra B Ra = Ra + EA(B)
3 address add Ra Rb C Ra = Rb + EA(C)

**Load/Store:**

3 address add Ra Rb Rc Ra = Rb + Rc
load Ra Rb Ra = mem[Rb]
store Ra Rb mem[Rb] = Ra

A load/store architecture has instructions that do either ALU operations or access memory, but never both.
Functionality

- calculate: \( A = X \times Y - B \times C \)

<table>
<thead>
<tr>
<th>SP</th>
<th>X</th>
<th>Y</th>
<th>B</th>
<th>C</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>+4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

stack | accumulator | register-memory | load-store
Functionality

- calculate: $A = X \times Y - B \times C$

| SP     | X   | SP 4 | +4 | Y   | SP 8 | +8 | B   | SP 12 | +12 | C   | SP 16 | +16 | A   |

Stack | Accumulator | Register-memory | Load-store

Push 8(SP)
Push 12(SP)
Mult
Push 0(SP)
Push 4(SP)
Mult
Sub
Store 16(SP)
Pop
Functionality

• calculate:  \[ A = X \times Y - B \times C \]
Functionality

- calculate: \( A = X \times Y - B \times C \)

```
SP  X
+4  Y
+8  B
+12 C
+16 A

stack          accumulator          register-memory          load-store
Push 8(SP)     Load 8(SP)            Mult R1 0(SP) 4(SP)      Pop
Push 12(SP)    Mult 12(SP)           Mult R2 8(SP) 12(SP)
Mult           Store 20(SP)          Sub 16(SP) R1 R2
Push 0(SP)     Load 4(SP)            
Push 4(SP)     Mult 0(SP)            
Mult           Sub 20(SP)            
Sub            Store 16(SP)          
Store 16(SP)   
Pop
```
Functionality

- calculate: \( A = X \times Y - B \times C \)

```
SP   X
+4   Y
+8   B
+12  C
+16  A

stack          accumulator          register-memory          load-store
Push 8(SP)     Load 8(SP)           Mult R1 0(SP) 4(SP)     Load R1 0(SP)
Push 12(SP)    Mult 12(SP)          Mult R2 8(SP) 12(SP)    Load R2 4(SP)
Mult           Store 20(SP)         Sub 16(SP) R1 R2           Load R3 8(SP)
Push 0(SP)     Load 4(SP)           Load R4 12(SP)           Load R4 12(SP)
Push 4(SP)     Mult 0(SP)           Mult R5 R1 R2            Mult R5 R1 R2
Mult           Sub 20(SP)           Mult R6 R3 R4            Mult R6 R3 R4
Sub            Store 16(SP)         Sub R7 R5 R6              Sub R7 R5 R6
Store 16(SP)   St 16(SP) R7         St 16(SP) R7              St 16(SP) R7
Pop
```
Trade-offs

- **Stack**
  - Short instructions
  - Lots of instructions
  - Simple hardware
  - Little exposed architecture

- **Accumulator**
  - See “stack”

- **Register-memory**
  - Expressive instructions
  - Few instruction
  - Instructions are complex and diverse
  - Lots of exposed architecture

- **Load-store**
  - Simple
  - Higher instruction count
  - Lots of exposed architecture
Memory Considerations

- *Effective Address* - memory address specified by the addressing mode
- How complex should the addressing modes be?
- What are the trade-offs?
Memory Considerations

- *Effective Address* - memory address specified by the addressing mode
- How complex should the addressing modes be?
- What are the trade-offs?
  - How widely applicable are they?
  - How much do they impact the complexity of the machine?
  - How many extra bits do they require to encode?
Instruction Operands

- **non-memory**
  - Register direct: Add R4, R3
  - Immediate: Add R4, #3

- **Memory**
  - Displacement: Add R4, 100 (R1)
  - Indirect: Add R4, (R1)
  - Indexed: Add R3, (R1 + R2)
  - Direct: Add R1, (1001)
  - Mem. indirect: Add R1, @(R3)
  - Autoincrement: Add R1, (R2)+
  - Autodecrement: Add R1, -(R2)
Addressing Mode Utilization

Conclusion?
Which Operations?

- **Arithmetic**
  - add, subtract, multiply, divide
- **Logical**
  - and, or, shift left, shift right
- **Data Transfer**
  - load word, store word
- **Control flow**
  - branch
  - PC-relative
    - displacement added to the program counter to get target address

Does it make sense to have more complex instructions?
- e.g., square root, mult-add, matrix multiply, cross product ...
Branch Decisions

• How is the destination of a branch specified? (how many bits?)

• How is the condition of the branch specified?

• What about indirect jumps?
Types of branches (control flow)

- conditional branch: `beq r1, r2, label`
- jump: `jmp label`
- procedure call: `call label`
- procedure return: `return`
Branch Conditions

• Condition Codes
  - Processor status bits are set as a side-effect of executed instructions or explicitly by a compare and/or test instruction
    Ex: sub r1, r2, r3
    bz label

• Condition Register
  Ex: cmp r1, r2, r3
    bgt r1, label

• Compare and Branch
  Ex: bgt r1, r2, label
Displacement Size

- Conclusions?
## Encoding of Instruction Set

<table>
<thead>
<tr>
<th>Operation and no. of operands</th>
<th>Address specifier 1</th>
<th>Address field 1</th>
<th>Address specifier</th>
<th>Address field</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) Variable (e.g., VAX, Intel 80x86)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address specifier 1</th>
<th>Address field 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(b) Fixed (e.g., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address specifier 1</th>
<th>Address specifier 2</th>
<th>Address field</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address specifier</th>
<th>Address field 1</th>
<th>Address field 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(c) Hybrid (e.g., IBM 360/70, MIPS16, Thumb, TI TMS320C54x)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Compiler/ISA Interaction

- Compiler is primary customer of ISA
- Features the compiler doesn’t use are wasted
- Register allocation is a huge contributor to performance
- Compiler-writer’s job is made easier when ISA has
  - regularity
  - primitives, not solutions
  - simple trade-offs
- Compiler wants
  - simplicity over power
System/Compiler/ISA Issues

• Parameter passing

  ABI ("Application Binary Interface")

• Accessing data
  – Stack
  – global

• I/O, Interrupts, Virtual Memory, …
Our Desired ISA

- Load-Store register arch
- Addressing modes
  - immediate (8-16 bits)
  - displacement (12-16 bits)
  - register indirect
- Support a reasonable number of operations
- Don’t use condition codes – (or support multiple of them ala PPC)
- Fixed instruction encoding/length for performance
- Regularity (several general-purpose registers)
MIPS64 instruction set architecture

- 32 64-bit general purpose registers
  - R0 is always equal to zero
- 32 floating point registers
- Data types
  - 8-, 16-, 32-, and 64-bit integers
  - 32-, and 64-bit floating point numbers
- Immediate and displacement addressing modes
  - Register indirect is a subset of displacement
- 32-bit fixed length instruction encoding
MIPS Instruction Format

I - type instruction

<table>
<thead>
<tr>
<th>6</th>
<th>5</th>
<th>5</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>rs1</td>
<td>rd</td>
<td>Immediate</td>
</tr>
</tbody>
</table>

- Encodes: Loads and stores of bytes, words, half words
- All immediates (rd ≠ rs1 op immediate)
- Conditional branch instructions (rs1 is register, rd unused)
- Jump register, jump and link register
  (rd = 0, rs = destination, immediate = 0)

R - type instruction

<table>
<thead>
<tr>
<th>6</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>rs1</td>
<td>rs2</td>
<td>rd</td>
<td>func</td>
</tr>
</tbody>
</table>

- Register–register ALU operations: rd ≠ rs1 func rs2
- Function encodes the data path operation: Add, Sub, ...
- Read/write special registers and moves

J - type instruction

<table>
<thead>
<tr>
<th>6</th>
<th>26</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>Offset added to PC</td>
</tr>
</tbody>
</table>

- Jump and jump and link
- Trap and return from exception
MIPS instructions

- Read on your own and become comfortable speaking MIPS

- **LD** R1, 1000(R2)  
  R1 gets memory[R2 + 1000]

- **DADDU** R1, R2, R3  
  R1 gets R2 + R3

- **DADDI** R1, R2, #53  
  R1 gets R2 + 53

- **JALR** R2  
  RA gets PC + 4; Jump to R2

- **JR** R3  
  Jump to R3

- **BEQZ** R5, label  
  If R5 == 0, jump to label (label is within displacement)
Very Long Instruction Words

- Each instruction word contains multiple operations
  - The semantics of the ISA say that they happen in parallel
  - The compiler can (and must) respect this constraint
VLIW Example

- **RISC code**
  - $s1 = 1; \:$s2 = 1, $s3 = 4
  - add $s2, $s1, $s3
  - sub $s5, $s2, $s3
  - Sub sees 5 $s2 = 5

- **VLIW instruction word** :
  - $s1 = 1; \:$s2 = 1, $s3 = 4
  - <add $s2, $s1, $s3; sub $s5, $s2, $s3>
  - sub sees $s1 = 1.
VLIW’s History

• VLIW has been around for a long time
  • It’s the simplest way to get ILP, because the burden of avoiding hazards lies completely with the compiler.
  • When hardware was expensive, this seemed like a good idea.
• However, the compiler problem is extremely hard.
  • There end up being lots of noops in the long instruction words.
• As a result, they have either
  • 1. met with limited commercial success as general purpose machines (many companies) or,
  • 2. Become very complicated in new and interesting ways (for instance, by providing special registers and instructions to eliminate branches), or
  • 3. Both 1 and 2 -- See the Itanium from intel.
A VLIW compiler must identify instructions that can execute in parallel and will execute under the same conditions.

The easy place to look is within a “basic block” (a region of code with no branches or branch targets).

Basic blocks are too small (3-10 instructions on average).
Trace Scheduling

• Profile to find the “hot” path through the code
• Treat the hot path (or “trace”) as a single basic block for scheduling
• Add fix up code for the cases when execution doesn’t follow the hot path.
Trace Scheduling

TRACE SCHEDULING LOOP-FREE CODE
Trace Scheduling is Hard

- Building sufficiently long traces is difficult.
- Loops
  - Unroll them.
- Function calls
  - inline them, if possible
  - Virtual functions, etc. make this hard.
- Getting the fix-up code is challenging.
- The hot path might not be so hot.
VLIW Today

- VLIW is alive and well in Digital signal processors
- They are simple and low power, which is important in embedded systems.
- DSPs run the same, very regular loops all the time, and VLIW machines are very good at that.
- It is worthwhile to hand-code these loops in assembly.
Beyond VLIW

- In RISC ISA dependence information is implicit in the use of register names.
- In VLIW some non-dependence information is explicit.
- You can add additional explicit information about dependences to the ISA.
- Dependence information is necessary if the microarchitecture is going to exploit parallelism.
  - In some cases it easier to provide that information explicitly -- searching for it in hardware is expensive.
  - But not all dependence information is available to the compiler.
    - Branch outcomes affect dependences.
    - Memory dependences are, in general, unknowable.
- We’ll see richer ISAs later in the quarter.
Key Points

• Modern ISA’s typically sacrifice power and flexibility for regularity and simplicity.
  – trade off code density for greater micro-architectural flexibility

• Instruction bits are extremely limited
  – particularly in a fixed-length instruction format

• Registers are critical to performance
  – we want lots of them, with few usage restrictions attached

• Displacement addressing mode handles the vast majority of memory reference needs.