Agenda

- MIPS ISA
- Sample Questions

Review

- Registers in MIPS
  - Inside processor
    use for frequently accessed data and fast calculation
  - How many registers in MIPS: 32
  - How large are them: 32 bit
  - How they work
    - Load values from memory into registers
    - Store result from register to memory

Review

- Registers in MIPS
  - C code

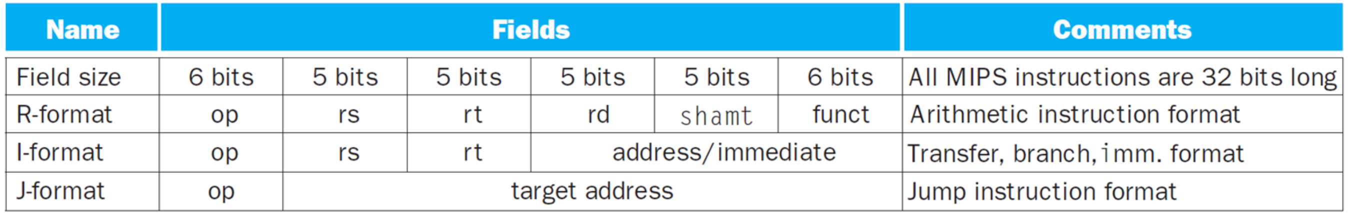
```
f = (g + h) - (i + j);
// g, h, i, j in $s1-s4
```

  - Compiled MIPS code

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

Review: Instructions in MIPS

- 3 simple formats
  - R-type, 3 register operands
  - I-type, 2 register operands and 16-bit immediate
  - J-type, 26-bit immediate operand



Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

- Simple decoding
  - 4 bytes per instruction, regardless of format
  - Must be 4-byte aligned
  - Format and fields readily extractable

ALU Instructions

- Assembly (e.g. register-register signed addition)
  - ADD rd_{reg} rs_{reg} rt_{reg}
- Machine encoding

0	rs	rt	rd	0	ADD
6-bit	5-bit	5-bit	5-bit	5-bit	6-bit

R-type

- Semantics
  - GPR[rd] ← GPR[rs] + GPR[rt]
  - PC ← PC + 4
- Exception on overflow
- Variations
  - Arithmetic: {signed, unsigned} x {ADD, SUB}
  - Logical: {AND, OR, XOR, NOR}
  - Shift: {Left, Right-Logical, Right-Arithmetic}

ALU Instructions

- Assembly (e.g. reg-immediate signed additions)
  - ADDI rt_{reg} rs_{reg} $immediate_{16}$
- Machine encoding

ADDI	rs	rt	immediate
6-bit	5-bit	5-bit	16-bit

I-type

- Semantics
  - GPR[rt] ← GPR[rs] + sign-extend (immediate)
  - PC ← PC + 4
- Exception on overflow
- Variations
  - Arithmetic: {signed, unsigned} x {ADD}
  - Logical: {AND, OR, XOR, LUI}

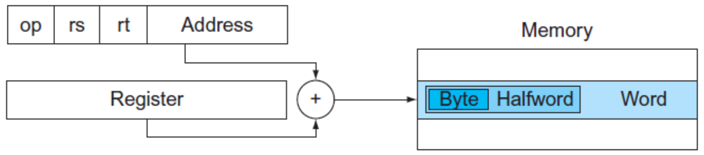
Load Instructions

- Assembly (e.g. load 4-byte word)
  - LW rt_{reg} $offset_{16}$ ($base_{reg}$)
- Machine encoding

LW	base	rt	offset
6-bit	5-bit	5-bit	16-bit

I-type

- Semantics
  - effective_address = sign-extend(offset) + GPR[base]
  - GPR[rt] ← MEM[translate(effective_address)]
  - PC ← PC + 4
- Exceptions
  - Address must be word-aligned
  - MMU exceptions



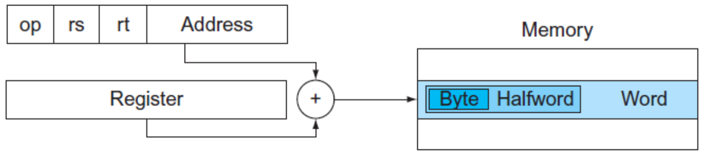
Store Instructions

- Assembly (e.g. store 4-byte word)
  - SW rt_{reg} $offset_{16}$ ($base_{reg}$)
- Machine encoding

SW	base	rt	offset
6-bit	5-bit	5-bit	16-bit

I-type

- Semantics
  - effective_address = sign-extend(offset) + GPR[base]
  - MEM[translate(effective_address)] ← GPR[rt]
  - PC ← PC + 4
- Exceptions
  - Address must be word-aligned
  - MMU exceptions



(Conditional) Branch Instructions

- Assembly (e.g. branch if equal)
  - BEQ rs_{reg} rt_{reg} $immediate_{16}$
- Machine encoding

BEQ	rs	rt	immediate
6-bit	5-bit	5-bit	16-bit

I-type

- Semantics
  - immediate: relative word address
  - branchAddr = sign-extend(immediate) x 4
  - if GPR[rs]==GPR[rt] then $PC \leftarrow PC + 4 + branchAddr$
    else $PC \leftarrow PC + 4$
- How far can you jump?
  - Within -2^15 to 2^15 words of the current instruction
- Variations
  - BEQ, BNE, BLEZ, BGTZ

Jump Instructions

- Assembly
  - J immediate$_{26}$
- Machine encoding

J	immediate
6-bit	26-bit

J-type

- Semantics
  - immediate: word address
  - jumpAddr = { PC + 4[31:28], immediate, 2’b0 }
  - PC $\leftarrow$ jumpAddr
- How far can you jump?
  - 2^26 words not related to the current instruction
- Variations
  - Jump and link
  - Jump registers

ISA-level Tradeoffs: Instruction Length

- **Fixed length**: Length of all instructions the same
  + Easier to decode single instruction in hardware
  + Easier to decode multiple instructions concurrently
  – Wasted bits in instructions (Why is this bad?)
  – Harder-to-extend ISA (how to add new instructions?)

- **Variable length**: Length of instructions different
  + Compact encoding (Why is this good?)
  – More logic to decode a single instruction
  – Harder to decode multiple instructions concurrently

- Tradeoffs
  – Code size (memory space, bandwidth, latency) vs. hardware complexity
  – ISA extensibility and expressiveness vs. hardware complexity
  – Performance? Energy? Smaller code vs. ease of decode

ISA-level Tradeoffs: Uniform Decode

- Uniform decode: Same bits in each instruction correspond to the same meaning
  - + Easier decode, simpler hardware
  - + Enables parallelism: generate target address before knowing the instruction is a branch
  - – Restricts instruction format or wastes space
  - e.g. MIPS, SPARC, Alpha

- Non-uniform decode
  - + More compact and powerful instruction format
  - – More complex decode logic
  - e.g. opcode can be the 1st – 7th byte in x86

ISA-level Tradeoffs: Number of Registers

- Affects
  - Number of bits used for encoding register address
  - Number of values kept in fast storage (register file)
  - Size, access time, power consumption of register file

- Large number of registers:

  + enables better register allocation (and optimization) by compiler -> fewer saves/restores

  – larger instruction size

  – larger register file size

ISA-level Tradeoffs: Addressing Modes

- Addressing mode specifies how to obtain an operand of an instruction
  - Register
  - Immediate
  - **Memory** (displacement, register indirect, indexed, absolute, memory indirect, autoincrement, autodecrement, ...)

- More modes:
  - + help better support programming constructs (array, pointer based accesses)
  - – make it harder for the architect to design
  - – many ways to do the same thing complicates compiler design

Other Example ISA-level Tradeoffs

- VLIW vs. single instruction
- Precise vs. imprecise exceptions
- Virtual memory vs. not
- Unaligned access vs. not
- Hardware interlocks vs. software-guaranteed interlocking
- Software vs. hardware managed page fault handling
- Cache coherence (hardware vs. software)
- …

RISC vs. CISC

	Format	Operations	Operands
RISC	Fixed length instructions Relatively simple encoding ARM/MIPS: 4B long	Simple, single function ops Single cycle	Operands: reg, imm Few addressing modes
CISC	Variable length instructions Common insts shorter/simpler Special insts longer/complex x86: from 1B to 16B long	Complex, multi-cycle insts	Operands: mem, reg, imm Many addressing modes

- RISC vs. CISC
  - While type of ISA has a better performance?
  - Is RISC more energy-efficient?

RISC vs. CISC

- Blem, Menon and Sankaralingam, "Power Struggles: Revisiting the RISC vs. CISC Debate on Contemporary ARM and x86 Architectures", HPCA 2013.
  - 11 key findings

- Conclusion: ISA being RISC or CISC does not matter for power and performance of modern processors

Sample Question 1

- C code:

```
int sum = 0;
 while (b != 0) {
     sum += a;
     b--;
 }
 sum = sum + 100;
```

- MIPS code?

Sample Question 1

```
        add   $t0, $zero, $zero
loop:   beq   $a1, $zero, finish
        add   $t0, $t0, $a0
        addi  $a1, $a1, -1
        j     loop
finish: addi  $t0, $t0, 100
```

Sample Question 2

Initially, $s3, $s4, $s5 contains i, j, k. Let $s6 stores the base of A[]. Each element of A is a 32-bit word.

MIPS instructions:

```
loop: add $t1, $s3, $s3
      add $t1, $t1, $t1
      add $t1, $t1, $s6
      lw  $t0, 0($t1)
      bne $t0, $s5, exit
      add $s3, $s3, $s4
      j loop
exit:
```

Sample Question 2

```
loop: add $t1, $s3, $s3
      add $t1, $t1, $t1
      add $t1, $t1, $s6
      lw  $t0, 0($t1)
      add $s3, $s3, $s4
      bne $t0, $s5, exit
      j loop
exit:
```

```
loop: t1 = 2i
      t1 = 4i
      t1 = s6 + 4i
      t0 = A[i]
      i = i + j
      if (A[i] != k) goto exit
      goto loop
exit:
```

```
while (A[i] == k) {
  i = i + j;
}
```

Slide credits

- Attiano Purpura-Pontoniere
- Yuchen Hao
- Onur Mutlu
- James C. Hoe