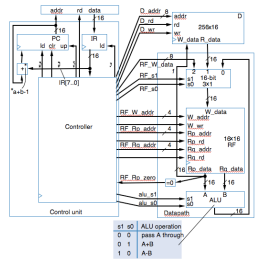
Lecture 07
Introduction to the MIPS ISA

Motivation: Why MIPS?

- Shortcomings of the simple processor
  - Only 16 bits for data and instruction
  - Data range can be too small
  - Addressable memory is small
  - Only support at most 16 instructions



- MIPS ISA: 32-bit RISC processor
  - A representative RISC ISA
    - (RISC – Reduced Instruction Set Computer)
  - A fixed-length, regularly encoded instruction set and uses a load/store data model
  - Used by NEC, Cisco, Silicon Graphics, Sony, Nintendo

MIPS vs. The 6-instruction Processor

A quick look: more complex ISAs

Instruction Encoding

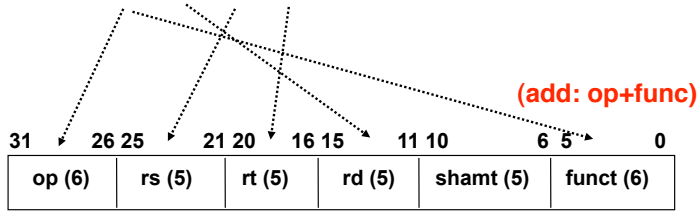
❑ 6-instruction processor:

Add instruction: **0010 $ra_3ra_2ra_1ra_0$ $rb_3rb_2rb_1rb_0$ $rc_3rc_2rc_1rc_0$**

Add Ra, Rb, Rc—specifies the operation $RF[a]=RF[b] + RF[c]$

❑ MIPS processor:

Assembly: add $9, $7, $8 # add rd, rs, rt: RF[rd] = RF[rs]+RF[rt]



Machine:

B:	000000	00111	01000	01001	xxxxx	100000
D:	0	7	8	9	x	32

A quick look: more complex ISAs

Instruction Encoding

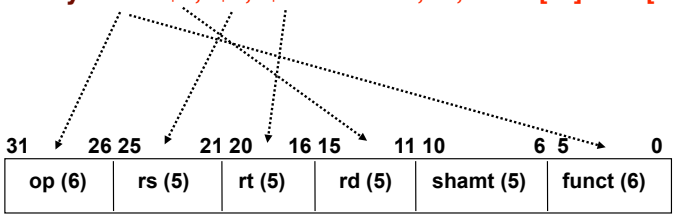
- 6-instruction processor:

Sub instruction: **0010 $ra_3ra_2ra_1ra_0$ $rb_3rb_2rb_1rb_0$ $rc_3rc_2rc_1rc_0$**

SUB Ra, Rb, Rc—specifies the operation *RF[a]=RF[b] – RF[c]*

- A MIPS subtract

Assembly: sub $9, $7, $8 # sub rd, rs, rt: RF[rd] = RF[rs]-RF[rt]

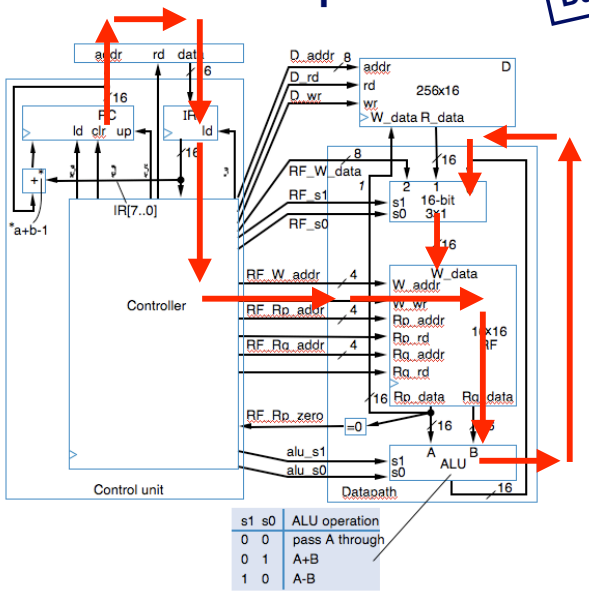


Machine:

B:	000000	00111	01000	01001	xxxxx	100010
D:	0	7	8	9	x	34

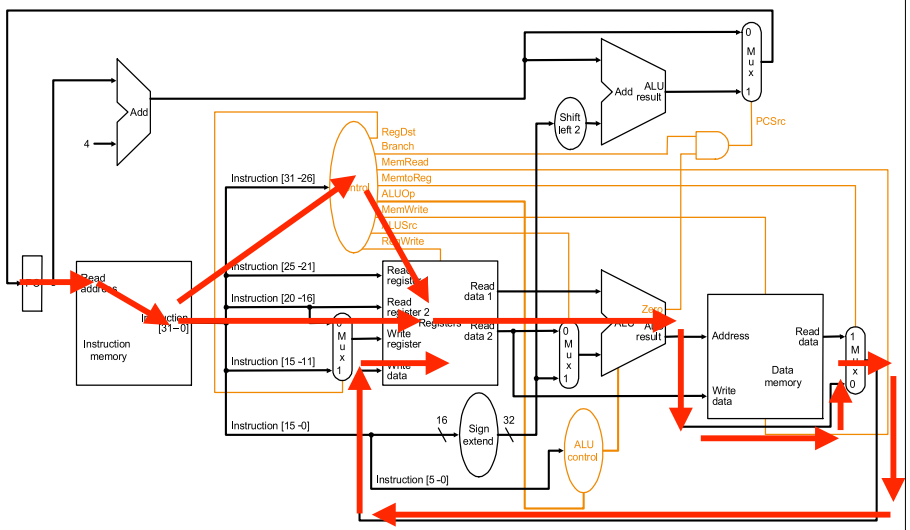
A quick look: more complex ISAs

Datapath



A quick look: more complex ISAs

Datapath



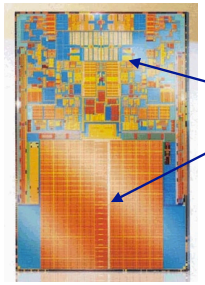
In terms of course work:

- In class and in homework assignments, we look at design issues that relate to modern machines
- In labs, we apply these ideas on a smaller scale (i.e. the 6-instruction processor) and tie lessons learned in the lab back to class work
- Before we talk more about MIPS, let's spend a few slides thinking about how this fits into the big picture.

Back to course goals...

- At the end of the semester, you should be able to...
  - ...describe the fundamental components required in a single core of a modern microprocessor
    - ~~(Also, explain how they interact with each other, with main memory, and with external storage media...)~~

Example



How do on-chip memory, processor logic, main memory, disk interact?

2.0 GB

$200.00

Apple Memory Module 2GB 667MHz DDR2 (PC2-5300) 2x1GB SO-DIMMs

Estimated Ship: Within 24 hours

Free Shipping

750GB SATA Hard Disk Drive Kit for...

Ships: Within 24hrs

Free Shipping

★★★★★

$299.00

Back to course goals...

- At the end of the semester, you should be able to...
  - ...understand how code written in a high-level language (e.g. C) is eventually executed on-chip...

Example

In C:

```
void insertionSort(int numbers[], int array_size)
{
  int i, j, index;

  for (i=1; i < array_size; i++)
  {
    index = numbers[i];
    j = i;
    while ((j > 0) && (numbers[j-1] > index))
    {
      numbers[j] = numbers[j-1];
      j = j - 1;
    }
    numbers[j] = index;
  }
}
```

See chalk board.

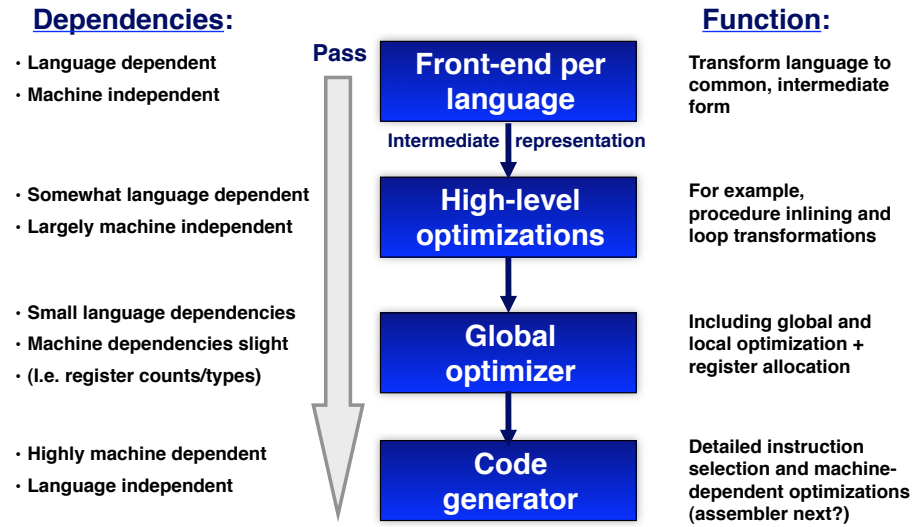
In Java:

```
public static void insertionSort(int[] list, int length) {
    int firstOutOfOrder, location, temp;

    for(firstOutOfOrder = 1; firstOutOfOrder < length; firstOutOfOrder++) {
       if(list[firstOutOfOrder] < list[firstOutOfOrder - 1]) {
          temp = list[firstOutOfOrder];
          location = firstOutOfOrder;

          do {
             list[location] = list[location-1];
             location--;
          }
          while (location > 0 && list[location-1] > temp);

          list[location] = temp;
       }
    }
}
```

Both programs could be run on the same processor... how does this happen?

That said...

A reason today's compilers work like this:



We'll discuss MIPS more in a bit...
...but 1st, a few slides on ISAs in general.

Instructions Sets

- "Instruction set architecture is the structure of a computer that a machine language programmer (or a compiler) must understand to write a correct (timing independent) program for that machine"
  - IBM introducing 360 (1964)
- an instruction set specifies a processor's functionality
  - what operations it supports
  - what storage mechanisms it has & how they are accessed
  - how the programmer/compiler communicates programs to processor

ISA = "interface" between HLL and HW

...

ISAs may have different sytnax (6-instruction vs. MIPS), but can still support the same general types of operations (i.e. Reg-Reg)

Instruction Set Architecture

- Must have instructions that
  - Access memory (read and write)
  - Perform ALU operations (add, multiply, etc.)
  - Implement control flow (jump, branch, etc.)
    - I.e. to take you back to the beginning of a loop
- Largest difference is in accessing memory
  - Operand location
    - (stack, memory, register)
  - Addressing modes
    - (computing memory addresses)
      - (Let's digress on the board and preview how MIPS does a load)
      - (Compare to 6-instruction processor?)

What makes a good instruction set

- implementability
  - supports a (performance/cost) range of implementations
    - implies support for high performance implementations
- programmability — A bit more on this one...
  - easy to express programs (for human and/or compiler)
- backward/forward compatibility
  - implementability & programmability across generations
    - e.g., x86 generations: 8086, 286, 386, 486, Pentium, Pentium II, Pentium III, Pentium 4...
- think about these issues as we discuss aspects of ISAs

Programmability

- a history of programmability
  - pre - 1975: most code was hand-assembled
  - 1975 – 1985: most code was compiled
    - but people thought that hand-assembled code was superior
  - 1985 – present: most code was compiled
    - and compiled code was at least as good as hand-assembly

over time, a big shift in what "programmability" means

Today's Semantic Gap

- popular argument: today's ISAs are targeted to one HLL, and it just so happens that this HLL (C) is very low-level (assembly++)
  - i.e. i = j + k; vs. Add Ri, Rj, Rk
  - would ISAs be different if Java was dominant?
    - more object oriented?
    - support for garbage collection (GC)?
    - support for bounds-checking?
    - security support?

Instruction Set Aspects

- #1 format
  - length, encoding
- #2 operations
  - operations, data types, number & kind of operands
- #3 storage
  - internal: accumulator, stack, general-purpose register
  - memory: address size, addressing modes, alignments
- #4 control
  - branch conditions, special support for procedures, predication

Aspect #1: Instruction Format

- fixed length (most common: 32-bits)
  - (plus) easy for pipelining (e.g. overlap) and for multiple issue (superscalar)
    - don't have to decode current instruction to find next instruction
  - (minus) not compact
    - Does the MIPS add "waste" bits?
- variable length
  - (plus) more compact
  - (minus) hard (but do-able) to superscalarize/pipeline
    - PC = PC + ???

Variable Addressing Mode

- Variable addressing mode – allows virtually all addressing modes with all operations
  - Best when many addressing modes & operations

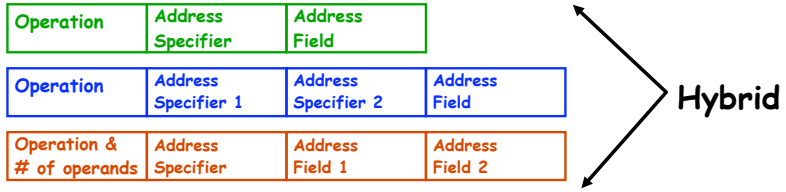
i.e. register-memory, memory-memory, register-register... all possible

Some random comments

- Variable addressing mode – allows virtually all addressing modes with all operations i.e. register-memory, memory-memory, register-register...
  - Best when many addressing modes & operations
- Fixed addressing mode – combines operation & addressing mode into opcode
  - Best when few addressing modes and operations
  - Good for RISC What's RISC? "Primitives not solutions." This is us.
-

Some random comments

- Variable addressing mode – allows virtually all addressing modes with all operations
  - Best when many addressing modes & operations
- Fixed addressing mode – combines operation & addressing mode into opcode
  - Best when few addressing modes and operations
  - Good for RISC What's RISC? "Primitives not solutions." This is us.
- Hybrid approach is 3rd alternative
  - Usually need a separate address specifier per operand



Aspect #2: Operations

- arithmetic and logical:
  - add, mult, and, or, xor, not
- data transfer:
  - move, load, store
- control:
  - conditional branch, jump, call, return
- system:
  - syscall, traps
- floating point:
  - add, mul, div, sqrt
- decimal:
  - addd, convert (not common today)
- string:
  - move, compare (also not common today)
- multimedia:
  - e.g., Intel MMX/SSE and Sun VIS
- vector:
  - arithmetic/data transfer, but on vectors of data

If no instruction for HLL operation, can "fake it" -- i.e. lots of adds instead of multiply.

Examples...

Data Sizes and Types

- fixed point (integer) data
  - 8-bit (byte), 16-bit (half), 32-bit (word), 64-bit (double)
- floating point data
  - 32/64 bit (IEEE754 single/double precision)
  - 80-bit (Intel proprietary)
- address size (aka "machine size")
  - e.g., 32-bit machine means addresses are 32-bits
  - virtual memory size key: 32-bits –> 4GB (not enough)
  - famous lesson:
    - one of the few big mistakes in an architecture is not enabling a large enough address space

Aspect #3: Internal Storage Model

- choices
  - stack
  - accumulator
  - memory-memory
  - register-memory
  - register-register (also called "load/store")
- running example:
  - add C, A, B (C := A + B)

Also this evolution b/c of evolution in HW complexity

Storage Model: Stack

```
push A   S[++TOS] = M[A];
push B   S[++TOS] = M[B];
add      T1=S[TOS--]; T2=S[TOS--]; S[++TOS]=T1+T2;
pop C    M[C] = S[TOS--];
```

- operands implicitly on top-of-stack (TOS)
- ALU operations have zero explicit operands
  - (plus) code density (top of stack implicit)
  - (minus) memory, pipelining bottlenecks (why?)
- mostly 1960's & 70's
  - x86 uses stack model for FP
    - (bad backward compatibility problem)
  - JAVA bytecodes also use stack model

Storage Model: Accumulator

```
load A   accum = M[A];
add B    accum += M[B];
store C  M[C] = accum;
```

- acc is implicit destination/source in all instructions
- ALU operations have one operand
  - (plus) less hardware, better code density (acc implicit)
  - (minus) memory bottleneck
- mostly pre-1960's
  - examples: UNIVAC, CRAY
  - x86 (IA32) uses extended accumulator for integer code

Storage Model: Memory-Memory

```
add C,A,B   M[C] = M[A] + M[B];
```

- no registers
  - (plus) best code density (most compact)
    - Why? Total # of instructions smaller for one...
  - (minus) large variations in instruction lengths
  - (minus) large variations in work per-instruction
  - (minus) memory bottleneck
- no current machines support memory-memory

Storage Model: Memory-Register

```
load R1,A    R1 = M[A];
add R1,B     R1 = R1 + M[B];
store C,R1   M[C] = R1;
```

- like an explicit (extended) accumulator
  - (plus) can have several accumulators at a time
  - (plus) good code density, easy to decode instructions
- asymmetric operands, asymmetric work per instruction
- 70's and early 80's
  - IBM 360/370
  - Intel x86, Motorola 68K

Storage Model: Register-Register (Ld/St)

```
load R1,A     R1 = M[A];
load R2,B     R2 = M[B];
add R3,R1,R2  R3 = R1 + R2;
store C,R3    M[C] = R3;
```

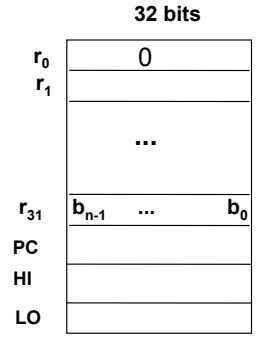
- load/store architecture: ALU operations on regs only
  - (minus) poor code density
  - (plus) easy decoding, operand symmetry
  - (plus) deterministic length ALU operations
  - (plus) fast decoding helps pipelining and superscalar
- 1960's and onwards
  - RISC machines: Alpha, MIPS, PowerPC (but also Cray)

On to MIPS

- MIPS is a register-register machine
- Aside from enhancements we made, 6-instruction is too!

MIPS Registers (R2000/R3000)

- 32x32-bit GPRs (General purpose registers)
  - $0 = $zero (therefore only 31 GPRs)
  - $1 = $at (reserved for assembler)
  - $2 - $3 = $v0 - $v1 (return values)
  - $4 - $7 = $a0 - $a3 (arguments)
  - $8 - $15 = $t0 - $t7 (temporaries)
  - $16 - $23 = $s0 - $s7 (saved)
  - $24 - $25 = $t8 - $t9 (more temporaries)
  - $26 - $27 = $k0 - $k1 (reserved for OS)
  - $28 = $gp (global pointer)
  - $29 = $sp (stack pointer)
  - $30 = $fp (frame pointer)
  - $31 = $ra (return address)



- 32x32-bit floating point registers (paired double precision)
- HI, LO, PC
- Status, Cause, BadVAddr, EPC

Board digression

- Programmer visibility
- Procedure calls

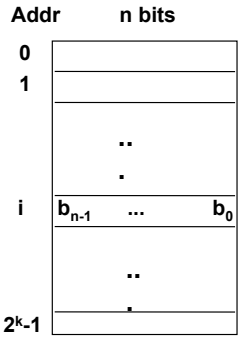
Memory Organization

- Addressable unit:
  - smallest number of consecutive bits (word length) can be accessed in a single operation
  - Example, n=8, byte addressable

Given 1K bit memory, 16 bit word addressable:

How many words?

How many address bits?

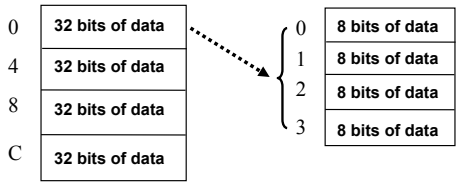


$(n*2^k)$ bits = $(n*2^{k-3})$ bytes

MIPS uses byte-addressable memory

Effect of Byte Addressing

MIPS: Most data items are contained in *words*,a word is 32 bits or 4 bytes. *Registers hold 32 bits of data*



- 2^{32} bytes with byte addresses from 0 to 2^{32}-1
- 2^{30} words with byte addresses 0, 4, 8, ... 2^{32}-4
- Words are aligned
- What are the least 2 significant bits of a word address?

A View from 10 Feet Above

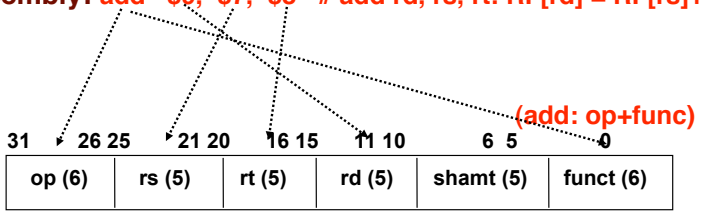
- Instructions are characterized into basic types
- Each type interpret a 32-bit instruction differently
- 3 types of instructions:
  - R type
  - I type
  - J type
- Look at both assembly and machine code

- In other words:
  - As seen with Add, instruction encoding broken down into X different fields
  - With MIPS, only 3 ways X # of bits arranged
    - Think about datapath: Why might this be good?

R-Type: Assembly and Machine Format

- R-type: All operands are in registers

Assembly: add $9, $7, $8 # add rd, rs, rt: RF[rd] = RF[rs]+RF[rt]



Machine:

B:	000000	00111	01000	01001	xxxxx	100000
D:	0	7	8	9	x	32

R-type Instructions

- All instructions have 3 operands
- All operands must be registers
- Operand order is fixed (destination first)
- Example:

C code: `A = B - C;`
(Assume that A, B, C are stored in registers s0, s1, s2.)

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

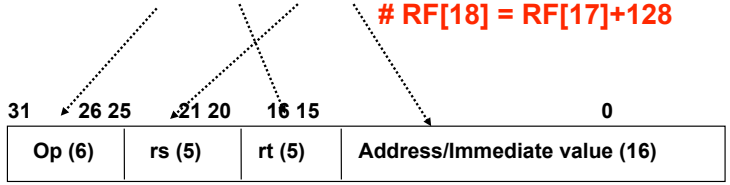
Machine code:

- Other R-type instructions
  - addu, mult, and, or, sll, srl, …

I-Type Instructions

- I-type: One operand is an immediate value and others are in registers

Example: addi $s2, $s1, 128 # addi rt, rs, Imm
RF[18] = RF[17]+128

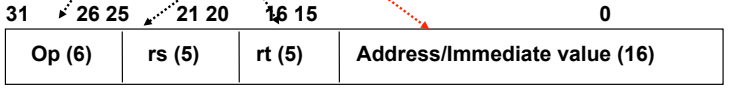


B:	001000	10001	10010	0000000010000000
D:	8	17	18	128

I-Type Instructions: Another Example

- I-type: One operand is an immediate value and others are in registers

Example: lw $s3, 32($t0) # RF[19] = DM[RF[8]+32]



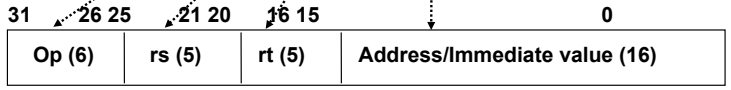
B:	100011	01000	10011	0000000000100000
D:	35	8	19	32

How about load the next word in memory?

I-Type Instructions: Yet Another Example

- I-type: One operand is an immediate value and others are in registers

Example: Again:: bne $t0, $t1, Again
if (RF[8]!=RF[9]) PC=PC+4+Imm*4
else PC=PC+4 (Why "4"?)



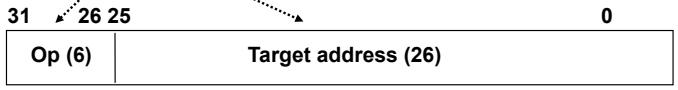
B: 00101 01000 01001 1111111111111111
D: 5 8 9 -1

PC-relative addressing

J-Type Instructions

- J-type: only one operand: the target address

Example: j 3 # PC = (PC+4)[31:28]||Target||00 (Why "00"?)



B: 000010 00000000000000000000000011
D: 2 3

Pseudo-direct Addressing

Example: Memory Access Instructions

- MIPS is a Load/Store Architecture (a hallmark of RISC)
  - Only load/store type instructions can access memory
- Example: A = B + C;
  - Assume: A, B, C are stored in memory, $s2, $s3, and $s4 contain the addresses of A, B and C, respectively.
    - lw $t0, 0($s3)
      - RF[8]=DM[RF[19]]
    - lw $t1, 0($s4)
      - RF[9]=DM[RF[20]]
    - add $t2, $t0, $t1
      - # RF[10]=RF[8]+RF[9]
    - sw $t2, 0($s2)
      - DM[RF[18]]=RF[10]
- sw has destination last
- What is the instruction type of sw?

See handout for lots of examples.