Lecture 06
Motivation and Background of MIPS ISA

Readings

- **Patterson and Hennessy:**
  - **Chapter 2**
    - **(Well, at least over the next 2 weeks...)**

Relatively speaking...

- **Let's illustrate how the 6-instruction processor fits into the grand scheme of modern computer architectures...**

A quick look: more complex ISAs

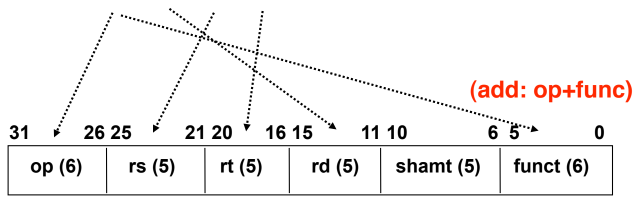
❑ **6-instruction processor:**

Add instruction: **0010 $ra_3ra_2ra_1ra_0$ $ra_3ra_2ra_1ra_0$ $ra_3ra_2ra_1ra_0$ $ra_3ra_2ra_1ra_0$**

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

- ❑ **6-instruction processor:**

Sub instruction: **0110 $ra_3ra_2ra_1ra_0$ $ra_3ra_2ra_1ra_0$ $ra_3ra_2ra_1ra_0$ $ra_3ra_2ra_1ra_0$**

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

- ❑ **A MIPS subtract**

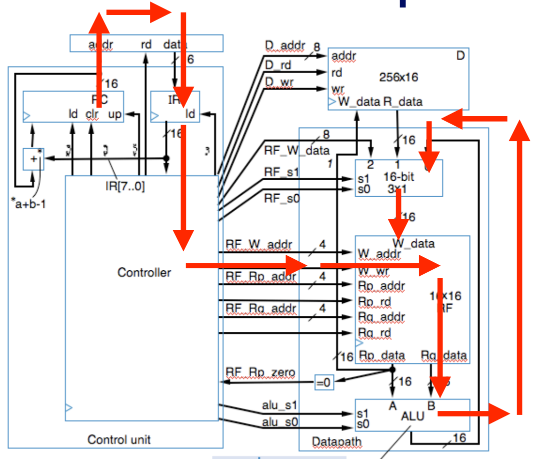
Assembly: sub $9, $7, $8 # sub rd, rs, rt: RF[rd] = RF[rs]-RF[rt]

31 26	25 21	20 16	15 11	10 6	5 0
op (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)

Machine:

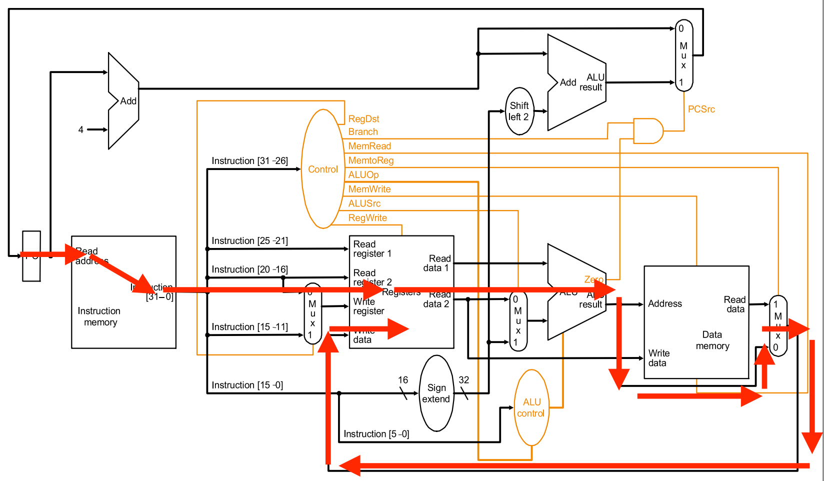
B:	000000	00111	01000	01001	xxxxx	100010
D:	0	7	8	9	x	34

A quick look: more complex ISAs



s1	s0	ALU operation
0	0	pass A through
0	1	A+B
1	0	A-B

A quick look: more complex ISAs



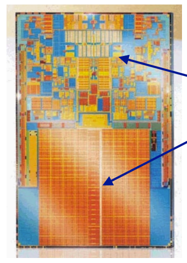
In terms of assignments:

- **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.**

So, what are the goals of this course?

- 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?

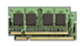
2.0 GB

$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

So, what are the goals of this course?

- 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;
  }
}
```

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?

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

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

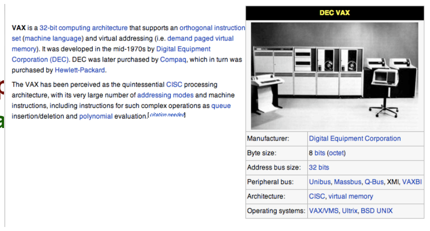
Programmability

- 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

pre-1975: Human Programmability

- focus: instruction sets that were easy for humans to program
  - ISA semantically close to high-level language (HLL)
    - closing the "semantic gap"
  - semantically heavy (CISC-like) instructions
    - automatic saves/restores on procedure calls
    - e.g., the VAX had instructions for polynomial evaluation
  - people thought computers would someday execute HLL directly
    - never materialized
  - one problem with this app
    - "semantic clash": not exa



VAX is a 32-bit computing architecture that supports an orthogonal instruction set (machine language) and virtual addressing (i.e. demand paged virtual memory). It was developed in the mid-1970s by Digital Equipment Corporation (DEC). DEC was later purchased by Compaq, which in turn was purchased by Hewlett-Packard.

The VAX has been perceived as the quintessential CISC processing architecture, with its very large number of addressing modes and machine instructions, including instructions for such complex operations as queue insertion/deletion and polynomial evaluation.[citation needed]

DEC VAX

Manufacturer:	Digital Equipment Corporation
Byte size:	8 bits (octet)
Address bus size:	32 bits
Peripheral bus:	Unibus, Massbus, Q-Bus, XMI, VAXBI
Architecture:	CISC, virtual memory
Operating systems:	VAX/VMS, Ultrix, BSD UNIX

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++)
  - would ISAs be different if Java was dominant?
    - more object oriented?
    - support for garbage collection (GC)?
    - support for bounds-checking?
    - security support?

A reason today's compilers work like this:

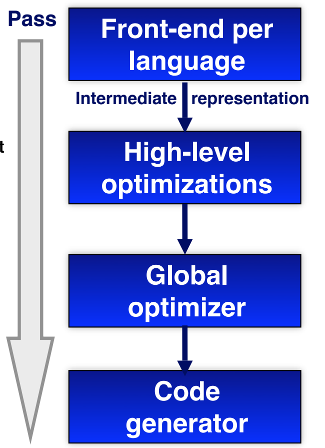
Dependencies:

- Language dependent
- Machine independent

- Somewhat language dependent
- Largely machine independent

- Small language dependencies
- Machine dependencies slight
- (I.e. register counts/types)

- Highly machine dependent
- Language independent



Function:

Transform language to common, intermediate form

For example, procedure inlining and loop transformations

Including global and local optimization + register allocation

Detailed instruction selection and machine-dependent optimizations (assembler next?)

Instruction Set Aspects

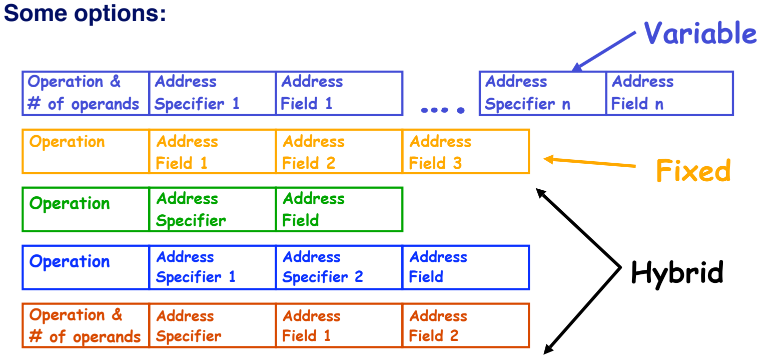
- #1 format
  - length, encoding
- #2 operations
  - operations, data types, number & kind of operands
- #3 storage
  - internal: accumulator, stack, general-purpose register
  - memory: address size, addressing modes, alignments
- #4 control
  - branch conditions, special support for procedures, predication

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 + ???

How is the operation specified?

- Typically in a bit field called the opcode
- Also must encode addressing modes, etc.
- Some options:


Variable
Operation & # of operands | Address Specifier 1 | Address Field 1 | ... | Address Specifier n | Address Field n
Fixed
Operation | Address Field 1 | Address Field 2 | Address Field 3
Hybrid
Operation | Address Specifier | Address Field
Operation | Address Specifier 1 | Address Specifier 2 | Address Field
Operation & # of operands | Address Specifier | Address Field 1 | Address Field 2

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

This is us.

- Hybrid approach is 3rd alternative
  - Usually need a separate address specifier per operand
- When encoding instructions, # of registers and addressing modes can affect instruction size

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)
    - especially since 1 bit is often used to distinguish I/O addresses
  - 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)

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!