Motivation

Fundamentally, transistor states can encode: a functional position, or a stored value interpreted as a number in binary (denoting a numeric value, a character shape, or a selector code perhaps for a specific sub-circuit or data pathway), see slide set 0.

Previously…

➪ **Physical device to (binary) code**

A machine needs to know what to do, in terms of its physical circuitry.

➪ **Specifying operations and operands**

Let's examine closely how a typical <u>modern</u> machine expects **operations** and **operands**, but first a major realization...

➪ **Logical representation (high-level) to machine specs**

Peeling the Layers

Computers are designed to completely hide the machine; we need to peel many layers.

Software

High Level Program

▶ (story starts here for most)

```
function listVerts() {
  for (var i=0; i<this.nv; i++) {
    var v = this.vert[i];
    print "VERT: ", i, v.label;
  }
}
```

Specify logical operations and vars (algorithm)*

*Example: repeatedly compare some orderable keys in certain ways (perhaps to sort).

Compile

Package + schedule requests according to machine specs

Encode requests for **operations** and pass **operands** in ways specific to machine.

machine code

```
00000000100000100001000000100000
10001100010011110000000000000000
10001100010100000000000000000100
...
```

Machine instr

Machine instr

Hardware

Decode

Unpack according to physical resources

Physical operations and operands

Repackage?

Perhaps the machine has enough circuitry to handle 3 at a time.

Machine

MIPS Register Operands
Basic Arithmetic

➪ Opcode

Fundamental to machine instructions, born out of need for speedy access. Number and width reflect a speed-power tradeoff driven by state of technology.

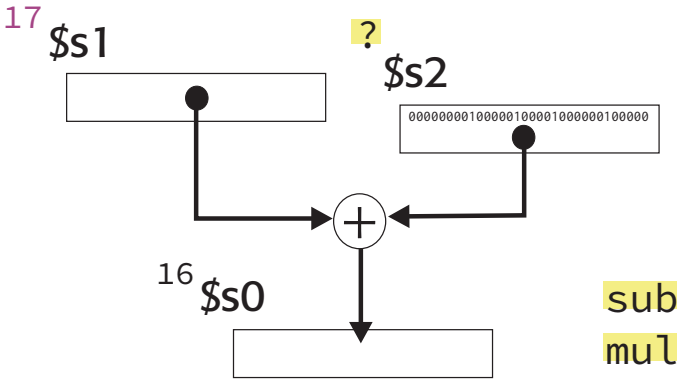
Original MIPS processor had 32 registers, denoted $0-$31, each 32 bits with fixed $0←0.

Registers `$16-$23` are reserved for high-level variables (referred to symbolically in MIPS assembly as `$s0-$s7`).

Assembly (language) is a symbolic representation of a binary machine instruction.

[Symbolic] Opcode

```
add $s0,$s1,$s2
```




```
sub $16,$17,$18
mult $16,$17,$18
div $16,$17,$18
```

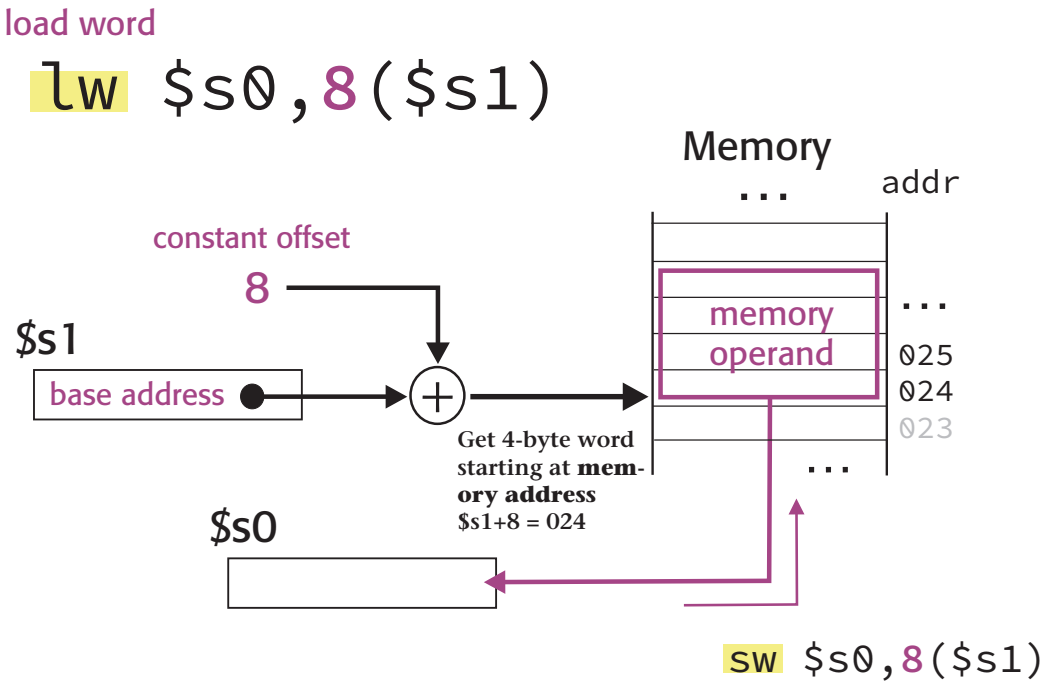
MIPS Memory Operands
Load-Store

A numbering scheme for logically usable memory cells (=**bytes**).

➪ **Memory address**

Specification

- Read a **base addr** from $s1
- Sum base addr and constant passed within instr (8)
- Fetch 4-byte data word using sum as memory addr
- Store data word in $s0



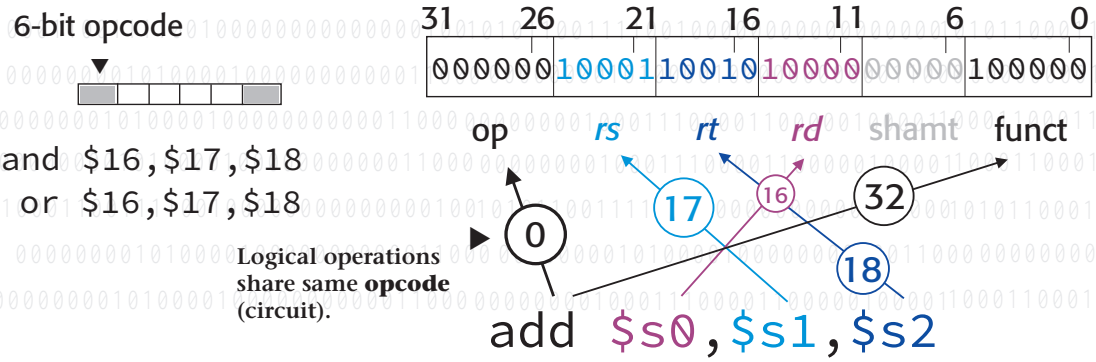
Machine Instructions

➪ Instruction word

```
 2 10
   11
  100
  101
  110
  111
 1000
 1001
10 1010
 1011
 1100
 1101
 1110
15 1111
10000
10001
18 10010
```

```
add  $2,$4,$2
lw   $15,0($2)
lw   $16,4($2)
sw   $16,0($2)
sw   $15,4($2)
```

```
00000000100000100001000000100000
10001100010011110000000000000000
10001100010100000000000000000100
10101100010100000000000000000000
10101100010011110000000000000100
```




```
and $16,$17,$18
 or $16,$17,$18
```

Logical operations share same **opcode** (circuit).

The machine sees a **stream** of bits, not neatly formatted instruction words.

Compilers decide how to sequence, i.e., schedule, instruction words hence the term **program**.

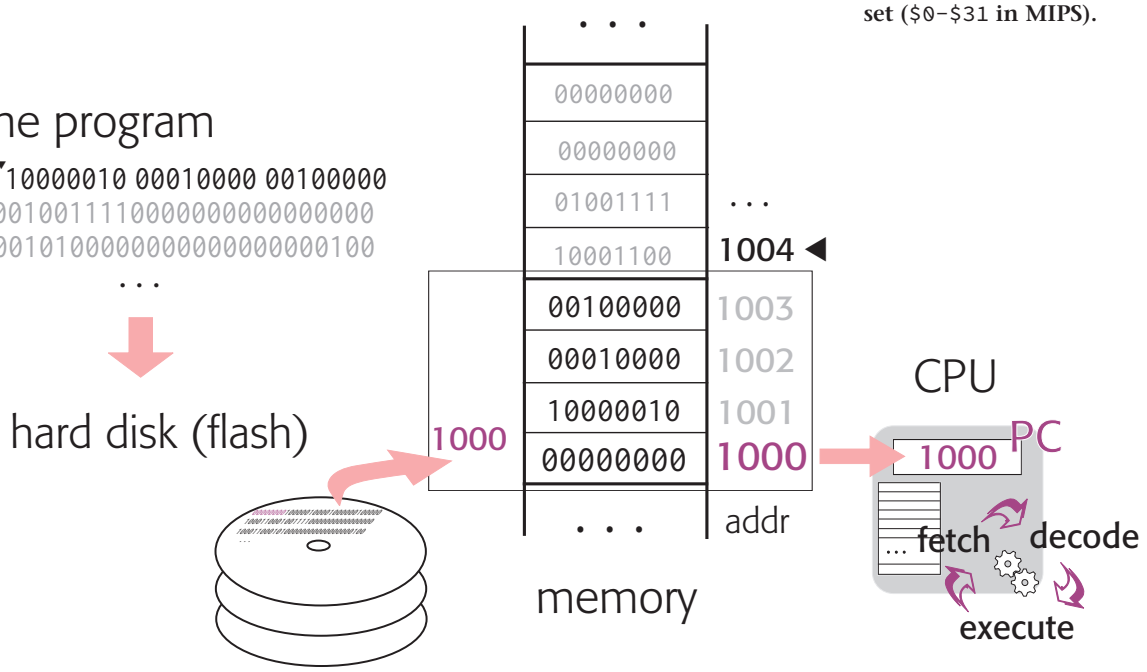
Program Execution

Program counter (PC)

A special register not part of the **general purpose register** (**GPR**) set (`$0-$31` in MIPS).

A **program** is a scheduled sequence of instruction words.

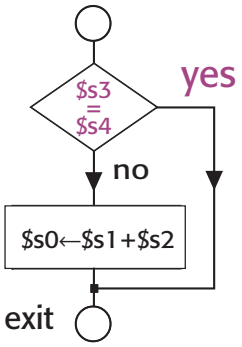
Quiz
Which component is responsible for finding a program on the hard disk (increasingly, flash/solid-state device), loading it in memory, and placing its start address in the PC?



Decision Instructions
Conditional Branch

➪ Memory label

Branch/Jump instructions allow programmers to conditionally or un-conditionally write to the PC.



Branch **taken** = PC written, breaking default execution flow.

Branch *taken* if equal

```
      beq $s3,$s4,EXIT
      add $s0,$s1,$s2
EXIT: ...
```

▲ Symbolic Address

Branch + logical instructions implement high-level conditional and loop statements.

Branch *taken* if not equal

```
      bne $s3,$s4,EXIT
      add $s0,$s1,$s2
EXIT: ...
```

Branch taken if greater-than 0

```
      bgtz $s0,EXIT
```

Set value on less-than condition

```
$s1 < $s2 ? 1: 0
slt $s0,$s1,$s2
```

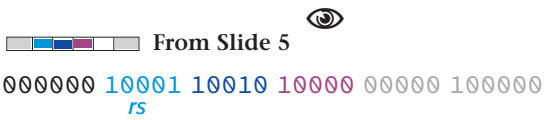
A memory label corresponds to var name in a high-level language.

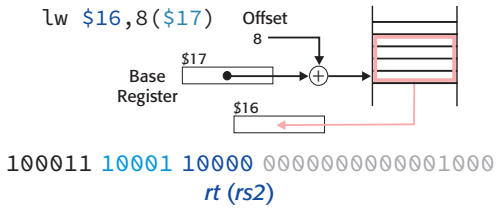
```
WHILE: beq $s3,$s4,EXIT
       add ...
       add $s3,$s3,1
     ▶ j   WHILE
EXIT: ...
```

Addressing Modes

Essentially ways to specify **operands**, an important aspect of machine instructions.

- ➪ **Immediate operand**
- ➪ **Load-store machine**

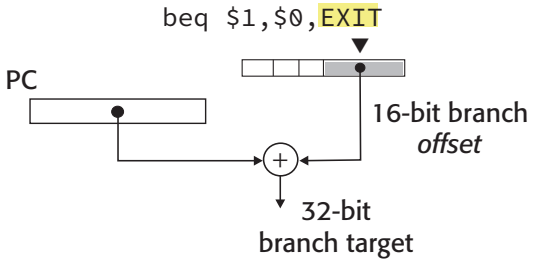




- ✓ **Register addressing**
- ✓ **Base (displacement)**

Immediate addressing

```
addi $16,$0,25000
```



PC-relative addr

Pseudodirect

Is it a good idea to provide an opcode to add a memory operand to a register operand? What would need to be changed about MIPS instr design?

Machine Operands Variations

Separate instructions deal with important operand variations and complications resulting from finite bit representations.

Quiz
How can a machine tell apart different operands? What are the consequences for a compiler? *Hint: physical machine's concerns may differ from those of programmers.*

Typically **2's complement** signed integers in as many bits as fits in a GPR, corresponding to `int` data type in C-like languages.

⇨ **Default operands**

⇨ **Unsigned integers**

Engineering and scientific calculations require a versatile representation of real numbers.

⇨ **Floating point reals**

Byte/double-byte operands are important for processing text (ASCII and Unicode characters).

⇨ **Short operands**

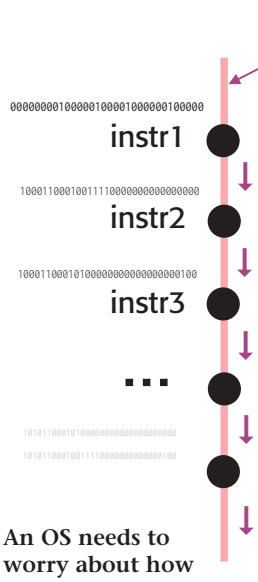
Shorter floating point operands optionally allow faster processing at expense of range and precision.

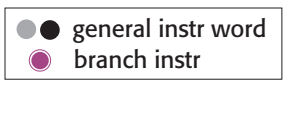
⇨ **Complications (overflow...)**

Answer Different opcodes are used for different operand types and related operations.

Software Execution Model

- Execution thread
- Control flow

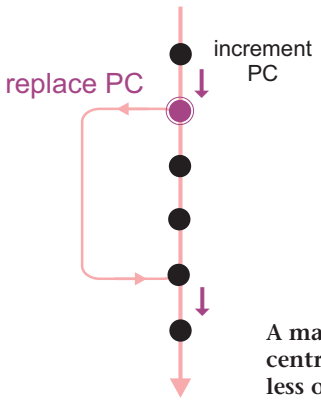




Default execution sequence provided by incrementing a PC may be visualized as a **thread** through consecutive instructions (sequencing control can be thought to flow along such threads).

An OS needs to worry about how instructions should be made to flow through hardware, not their specifics.

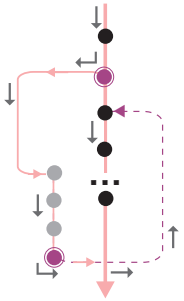
Control flow is an abstraction of physical sequencing performed by the processor's **control unit** (actual execution sequence may be more complex than what programmers see).



In reality some sequences are logically independent and can be performed in any order, or *threaded* concurrently.

A machine may provide a centralized control flow regardless of available execution threads. Extra resources are needed to run threads concurrently, at a minimum, an OS which allows threads to simulate concurrence (by transferring control around).

This control flow pattern(?) still describes a single execution thread.



Instruction Models

➪ Instruction set

Essentially a set of **opcodes** and **operand** specifications which the hardware can recognize.

Quiz
Why would the move from a **32-bit processor** to 64-bit be significant from instruction set viewpoint?

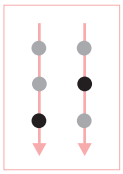
➪ CISC (Intel 8086): do more

➪ **RISC (MIPS): do less**

Essentially pack multiple ops in one instr to relieve fetch-decode burdens.

➪ **VLIW: do in parallel (later)**

Program execution



Multithreading can create apparent concurrency; programs will seem to run faster.

Exercise
Lookup the difference beween **parallelism** and **concurrency.**

- ✎ Dataflow model origins
- ✎ Where multi-threading fits?
- ✎ Hardware threads

If extra resources are available program threads can truly be concurrent.

Register Size Limitation

The number of bits internal registers can hold is a major architectural feature.

Historically, GPR (primarily for operands) were not used to hold addresses. MIPS re-uses GPR for addressing also.

⇨ Memory addressing

Limit directly addressable memory locations

Most machines will not provide circuitry to handle more bits than associated registers can store.

⇨ Computation

Limit computation performed directly in hardware

Reg size can influence instruction design and capabilities in subtle ways.

What if address or number don't fit? Breakdown computation in software

What Exactly is Software?

```
f = (g+h)-(i+j);
```

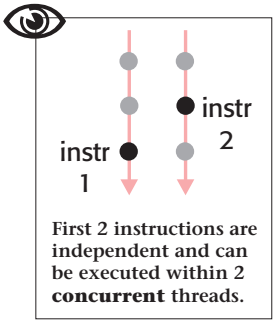
$s0–$s4 ← f, g, h, i, j
0 1 … 4

$t0←g+h, $t1←i+j

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

Most computers can add two 32-bit integers in physical registers using physical circuits, but can't calculate *f* directly since they lack a dedicated circuit.

A compiler can easily generate a sequence of machine instructions to perform the operation.



First 2 instructions are independent and can be executed within 2 **concurrent** threads.

Similarly, while classic MIPS (circa 1985) can't add two 128-bit integers in hardware, a small **program** (schedule of instructions) can do it.

The operation is said to be performed in **software** since no special circuitry is available for it.

$t1←g+h, $t2←i+j

```
add $t1,$s3,$s4      add $t1,$s1,$s2
add $t0,$s1,$s2      add $t2,$s3,$s4
sub $s0,$t0,$t1      sub $s0,$t1,$t2
```

Different machine programs (why?) even though they logically perform the same op.

A computer designer may decide to provide physical circuits for such expressions and a corresponding machine instruction though, as was the case often with CISC.

The Front-end

Typical scenarios (different machines implement variations depending on capabilities and design tradeoffs.)

Software (mostly)

Some ops performed by a machine instr directly in physical circuits (**hardware**).

Decode burden shared among compiler, a hardware decoder, and control unit.

Hardware (mostly)

Note the layers of processing?

Everything above this line is essentially overhead.

```
function listVerts() {
  for (var i=0; i<this.nv; i++) {
    var v = this.vert[i];
    print "VERT: ", i, v.label;  }}
```

Specify logical ops & operands

Compile ---------- Assembler instr

Compilers may provide soft instructions via an **assembler** to further hide true machine.

Instruction Set Architecture

Machine instr

Machine instr

Package + schedule according to machine specs

Front-end

Decode

Hardware ops

"Issue" individual or packets of hardware ops depending on machine resources (memory, functional units and paths)

Repackage?

Pass sequences of hard-ware command signals (**micro-code**).

Control Unit

Microarchitecture

Commit ("bind") Machine Resources & Execute

Note on Composability

Fused multiply-add

One instruction 2 jobs

```
r4 ← r1 + r2×r3
r1 ← r1 + r2×r3
```

Integer/floating-point versions and the 3-operand **multiply-accumulate** variant occur frequently in applications such as multi-media and digital signal processing which rely on matrix and vector math.

ARMv8 (RISC)

```
madd r4,r2,r3,r1
```

MIPS (R10000)

```
madd.d f4,f1,f2,f3
```

MIPS R2000/R3000

```
▸ madd $s4,$s1,$s2,$s3
    multu $s2,$s3
    mflo  $at
    add   $s4,$s1,$at
```

An assembler may provide a **pseudo-instruction** to be composed by a real machine sequence.

Indexed Addressing

Processing large arrays

```
$t1← mem[$a0+$s3]
```

Memory address obtained from 2 registers, one for array base address and the 2nd for an index (instead of a constant).

PowerPC (RISC)

```
lw $t1,$a0+$s3
```

8086/8088

```
MOV AX,[BX+SI]
```

MIPS R2000/R3000

```
addu $t0,$a0,$s3
lw   $t1,0($t0)
```

This version of add (opcode, really) is needed to treat operands as unsigned.

Simplified vs. Complex

Intel 8088 ADD

A memory operand can considerably complicate the execution profile of the instr on the same hardware.

Memory operand
9 cycles + 4 cycles (transfer)
+ address computation

Register operand
3 cycles

Conditional branch

Instruction set supports every potentially useful condition with a variety of operand scenarios.

- **Intel 8088/8086**

```
JG/JNLE JGE/JNL JL/JNGE
JLE/JNG JO JS JNO JNS
JA/JNBE JAE/JNB JB/JNAE
JBE/JNA
JC JE/JZ JP/JPE JNC JNE/JNZ
JNP/JPO
JCKZ CMP*
```

MIPS R2000/3000

```
beq bne bltz blez bgtz
bgez bltzal bgezal slt
slti sltu sltiu
```

MIPS supports a minimum necessary to compose the rest, favoring the more frequent scenarios, with fast register operands only.