

Fundamentally, transistors encode a switching state in a circuit interpreted as a number in binary, denoting a numeric value, a character shape, or a selector code for a sub-circuit.

Previously...

⇒ **Physical device to (binary) code: see Slide 0**

A computer, composed of physical devices that encode and transform a binary state (represented as bits), needs to know what to do in terms of its physical circuitry.

⇒ **Specifying operations and operands**

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

⇒ **From logical representation (high-level) to machine code**



An advanced review **not about MIPS or assembly programming**, focus on instruction issues, tradeoffs, and **examples**

Peeling the Layers



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

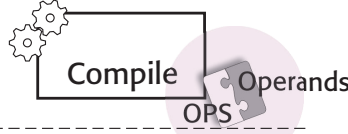
High Level Program
▶ (story starts here for most)

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

Specify logic/vars
(an algorithm)*

**Repeatedly compare orderable keys in certain ways (perhaps to sort), for example*

Software



Encode requested operations and pass operands in ways specific to machine

Machine code

```
10001100010011110000000000000000  
1000110001010000000000000000100
```



Machine instr



Machine instr

Package + schedule requests based on machine specs

Hardware



Unpack according to physical resources

Physical operations
& operands



Repackage?

machine may have enough circuitry to handle many ops at a time (3 in example)

Machine

MIPS Register Operands Basic Arithmetic

Register operands, fundamental to machine instructions, focus on speedy access (number and bit size reflect tradeoffs driven by the state of technology).

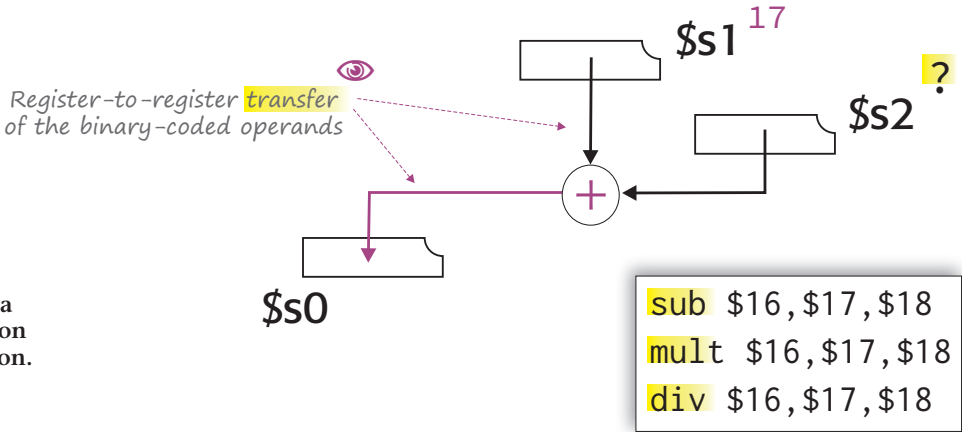
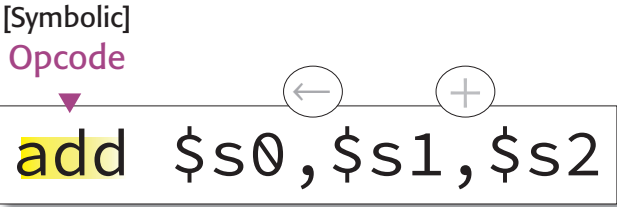
In 1985, the original MIPS processor had 32 **registers**, denoted **\$0-\$31**, each 32 bits with fixed **\$0** ← 0.

Registers **\$16-\$23** are symbolically referred to as **\$s0-\$s7** in MIPS **assembly**.

- ⇨ **Register, opcode**
- ⇨ **Assembly language**

8

An **Assembly instruction** is a human-symbolic representation of a binary machine instruction.



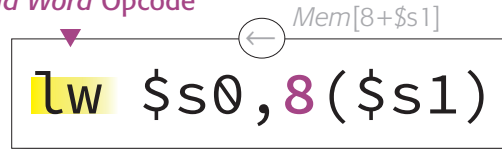
MIPS Memory Operands Load-Store

⇒ Memory address

A numbering scheme for logically usable memory cells, conventionally, every 8 bits (**byte**) given a unique numerical code.

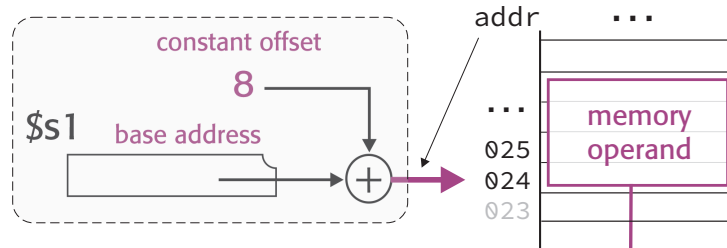
High-level variables are kept in memory, $s0-s7$ (\$16-23) used conventionally by the compiler to track up to 8 at a time.

Load Word Opcode



LW 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 $\$s0$



Get 4-byte word starting at memory byte address 024 to fill reg.



Register-memory transfer of binary-coded operands

$\$s0$

SW $\$s0, 8(\$s1)$

Machine Instructions

⇒ Instruction word

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

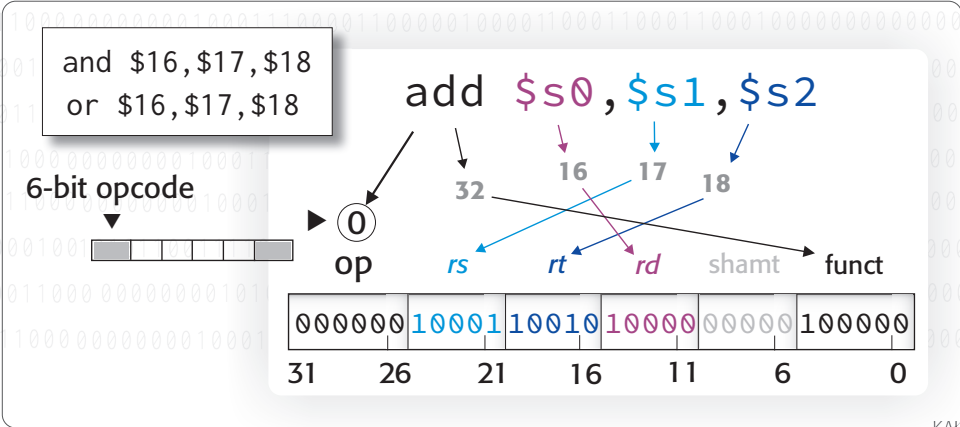
Quiz
Guess what the program does?

```
00000000100000100001000000100000 ▶ add $2,$4,$2
10001100010011110000000000000000   lw  $15,0($2)
10001100010100000000000000000000   lw  $16,4($2)
10101100010100000000000000000000   sw  $16,0($2)
10101100010100000000000000000000   sw  $15,4($2)
```

Logical operations share the same **opcode** (i.e., circuit, guess which one?).



The machine sees streams of bits, not neatly formatted instruction words.



Connections Program Execution

⇒ Program counter (PC)

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



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

Quiz

Which component is responsible for finding a program on the flash/hard disk, loading it in memory, and placing its start address in the PC?

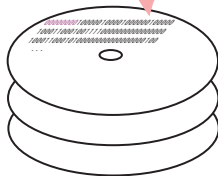
The classic 3-phase machine cycle matched actual processing in early CPU (nowadays generic, broken into more physical ones and typically involving multiple instructions).

Machine Program

00000000 10000010 00010000 00100000
10001100010011110000000000000000
1000110001010000000000000000100
...



Flash/hard Disk



1000

...	...
01001111	...
10001100	1004
00100000	1003
00010000	1002
10000010	1001
00000000	1000
...	addr

Memory

CPU

PC

1000

fetch
decode
execute

Decision Instructions

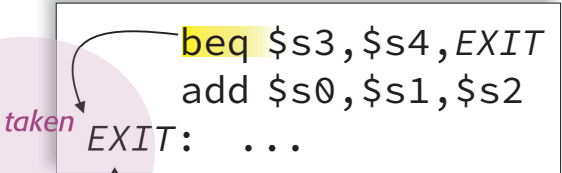
Conditional Branch

⇨ Memory label

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

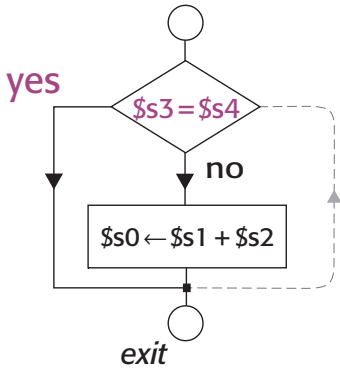
Branch **taken** = PC written, breaking the default execution flow.

Branch if equal



Symbolic Address

Branch + logical instructions implement high-level conditional and loop statements (a conditional is no more than a loop that runs zero or n=1 times).



Branch *taken* if not equal

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

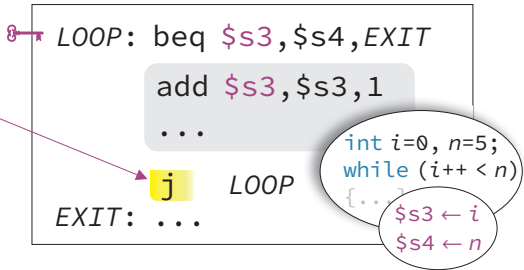
EXIT: ...

Branch taken if greater-than 0

```
bgtz $s0,EXIT
```

Set on less-than condition

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



Addressing Modes

- ⇒ Immediate operand
- ⇒ Load-store machine



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

- ✓ Register addressing
- ✓ Base (displacement)

Immediate addressing

addi \$16,\$0,25000

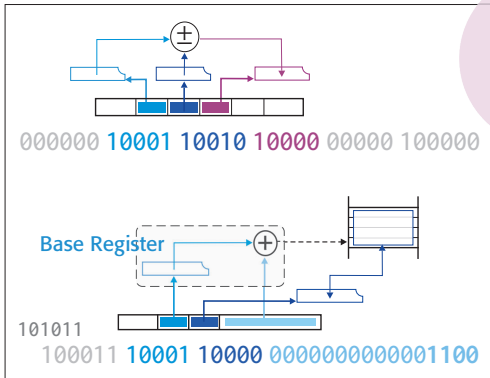
PC-relative addr

beq \$1,\$0,EXIT

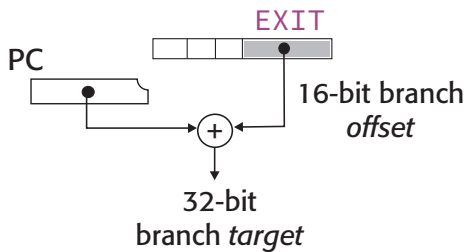


Is it a good idea to provide an opcode for adding a memory operand to a register operand? Specify changes to instruction design.

Pseudodirect



001000 0x61A8
25000



Machine Operands Variations

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

Typically 2's complement signed integers in as many bits as fits in a data register or GPR (in MIPS-like designs), 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...)**

Quiz

Different operands are indistinguishable bits for the machine, how can it tell them apart? What are the consequences for a compiler? *Hint: physical machine's concerns may differ from those of programmers (see Wulf).*

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

Software Execution Model

⇒ Execution thread

⇒ Control flow

An OS is concerned with how instructions flow through hardware, not their execution details.

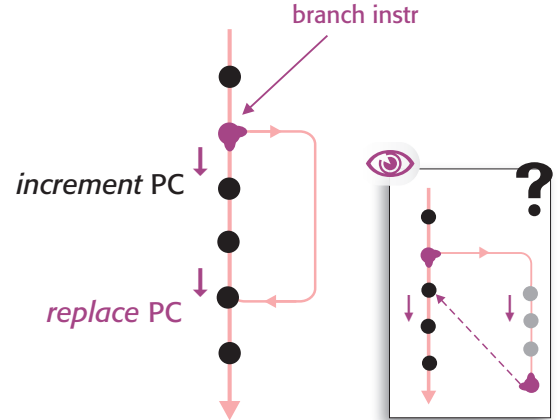
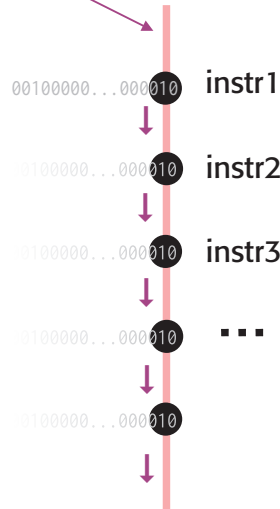
We may visualize the default execution sequence, due to incrementing a PC, as imaginary **thread** through consecutive instructions; **sequencing control** may be thought to **flow** along threaded instructions.

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

In practice, some instruction sequences from the same program are logically independent and may run in any order or *threaded* concurrently.

Extra resources are needed to run different threads concurrently. At a minimum, an OS that **simulates** concurrence by transferring control around.

Thread



Quiz

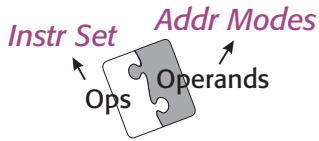
Compare a **procedure** to an **exception/interrupt**?

Hint: consider a scheduling viewpoint.

Instruction Models

⇒ Instruction set

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



⇒ CISC (Intel 8086): do more

⇒ RISC (MIPS): do less

⇒ VLIW: do in parallel (later)




Pack multiple independent ops in one instr to relieve fetch-decode burdens and facilitate parallel execution.

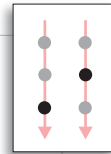
Natural **data flow** allowed concurrency where centralized control forced artificial sequencing, which motivated adding functional units to exploit it.

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

Exercise Compare parallelism and concurrency. 

Instruction execution

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



Register Size Limitation

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

Historically, separate registers were used for data operands, indexing, and addresses; MIPS uses GPR in all cases.

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

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



Old microprocessors had economic variants with narrower data bus (fewer pins) labeled 8/16-bit despite having the same wider internal registers (e.g., 8086/88, 68000/68008).

Quiz

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

⇒ **Memory addressing**
Limit directly accessible memory

⇒ **Computation**
Limit computation performed directly in hardware

What if operands don't fit?
Breakdown computation in *software*

What Exactly is Software?

Most computers can add two 32/64-bit integers in physical registers using physical circuits but can't calculate f directly since they mostly lack a dedicated circuit to perform a *difference-of-2-sums* operation.

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

$$f = (x+y) - (s+t);$$

12...5

\$s1-5 ← f,x,y,s,t

\$t1 ← x+y

\$t2 ← s+t

```
add $t1,$s2,$s3
add $t2,$s4,$s5
sub $s1,$t1,$t2
```

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 circuitry is dedicated for it.

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

add \$t2,\$s4,\$s5	add \$t0,\$s2,\$s3
add \$t1,\$s2,\$s3	add \$t1,\$s4,\$s5
sub \$s1,\$t1,\$t2	sub \$s1,\$t0,\$t1

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.

Typical scenarios, implementations vary depending on design tradeoffs.

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

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

Note layers of processing?

Everything above this line is essentially overhead.

The Front-end

High-level languages are the most significant front-end interface for humans.

High-level Language

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

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

Assembler "instructions" like move further hide the bare machine.

Specify logical ops & operands

Ops
Operands
Instructions

Software (mostly)

Front-end

Hardware (mostly)

Machine instr

Package + schedule based on ISA

Decode

Dependencies

Potentially reschedule

Hardware-ready ops/operands

Issue individual or packets of hardware ops depending on machine resources

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

Control

Multiple issue

Microarchitecture

Bare Machine

Bind (i.e., assign) Resources & Execute

Instruction Design 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.

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

- ▶ **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

Instruction Design

Note on Composability

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

A frequent op in significant applications (part of a sum-of-products calculations) for which a dedicated machine instruction (circuit) may be justified.

Fused multiply-add

One instruction, two jobs

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
```

Or, a machine may offer a **pseudo-instruction** (*gray*) to make programming easy, to be composed by a sequence of machine instructions (*in color*).

```
$t1 ← mem[$a0 + $s3]
```

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

Note specialized register for (base) indexing in 8086/8088 vs. GPR in RISC.

Indexed Addressing

Processing large arrays

PowerPC (RISC)

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

8086/8088 (CISC)

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

MIPS R2000/R3000

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

Note version (opcode) of add needed to treat operands as unsigned since dealing with an address.