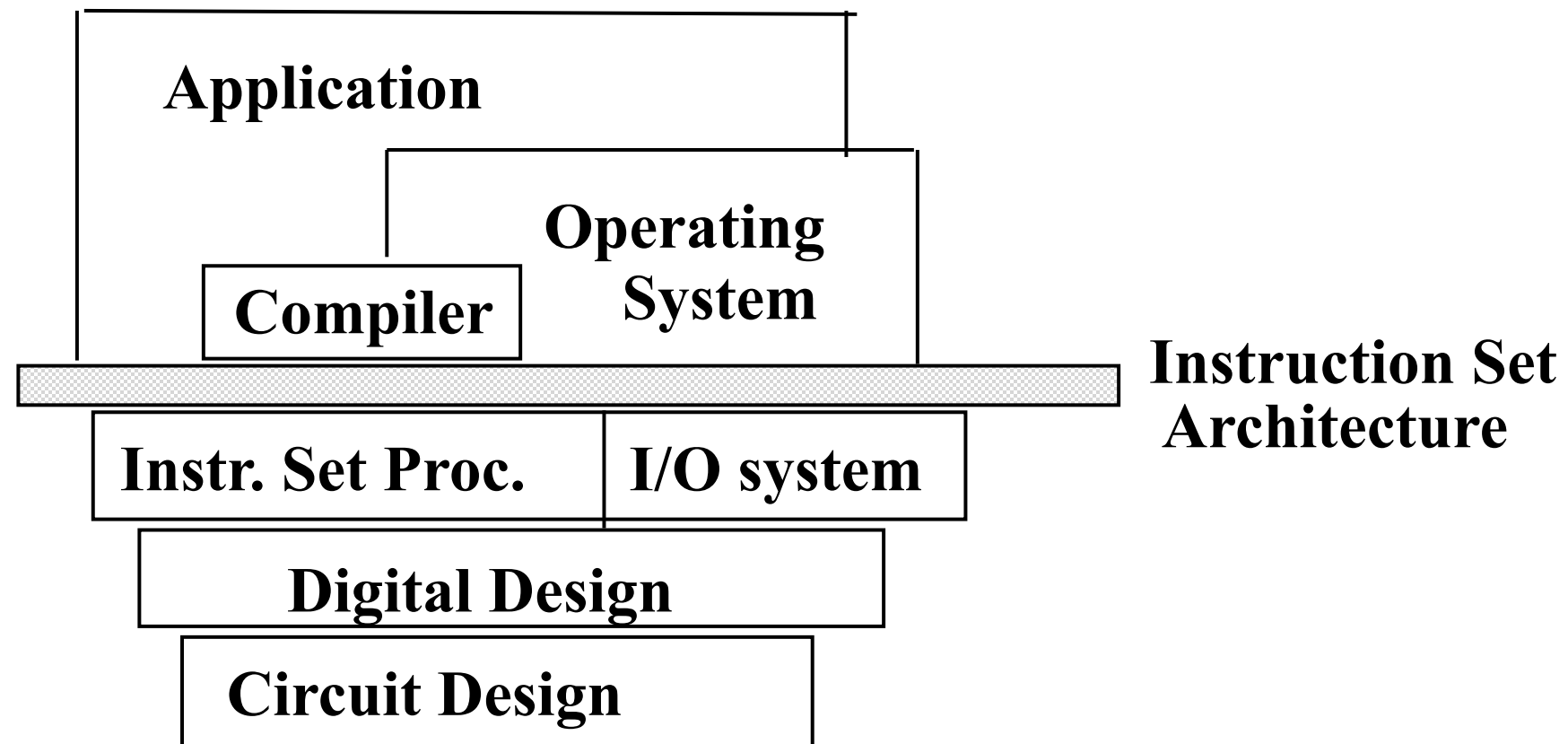


Instruction Set Architecture

Jason Mars

Instruction Set Architecture

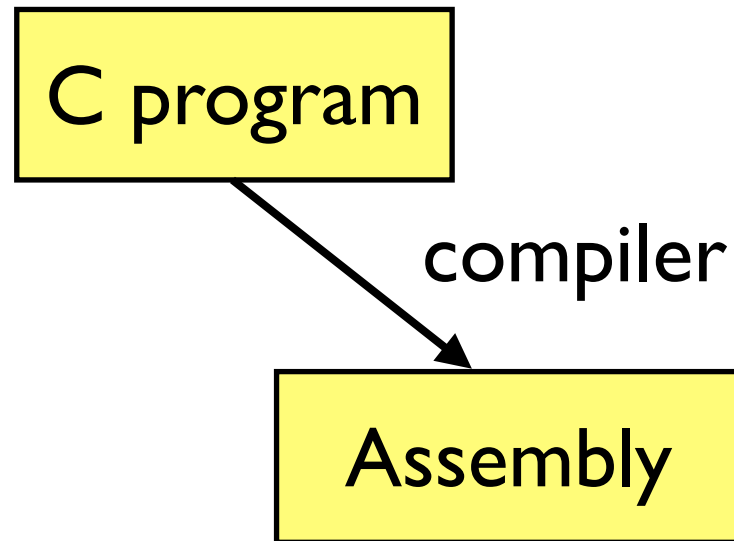
- The agreed-upon interface between all the software that runs on the machine and the hardware that executes it.



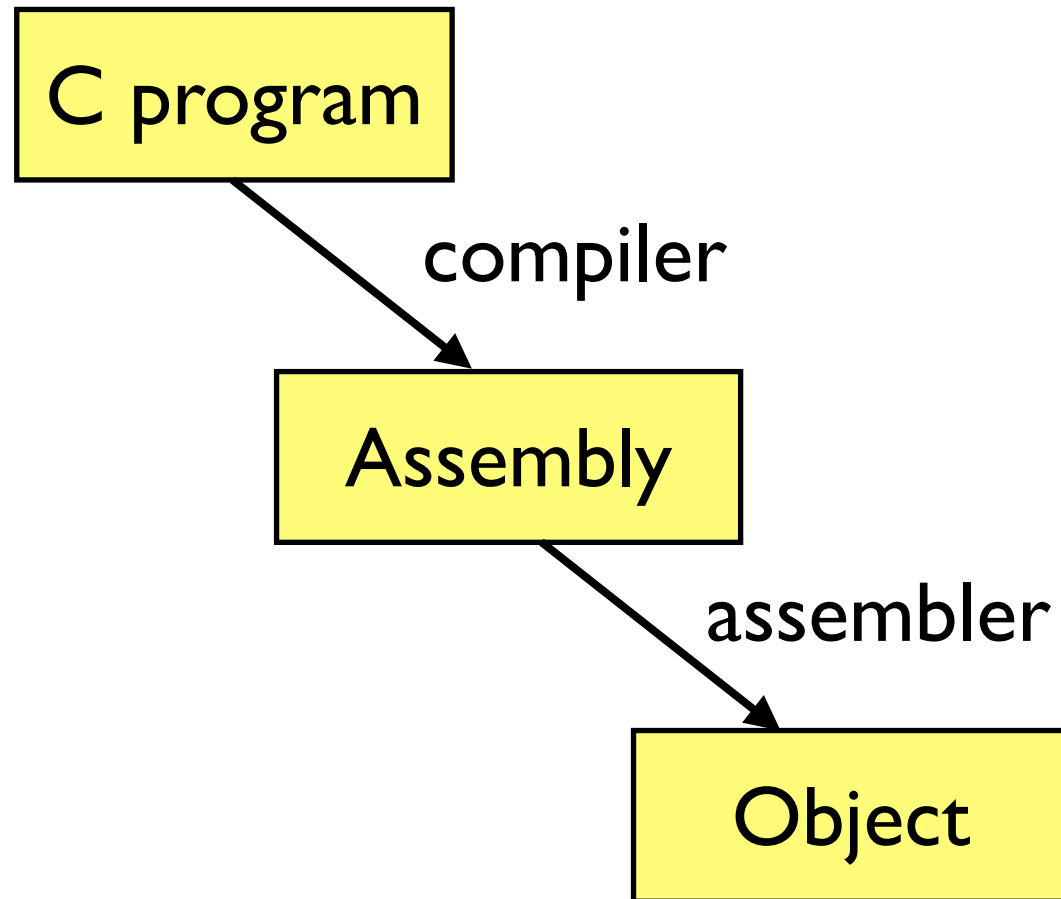
C to Assembly

C program

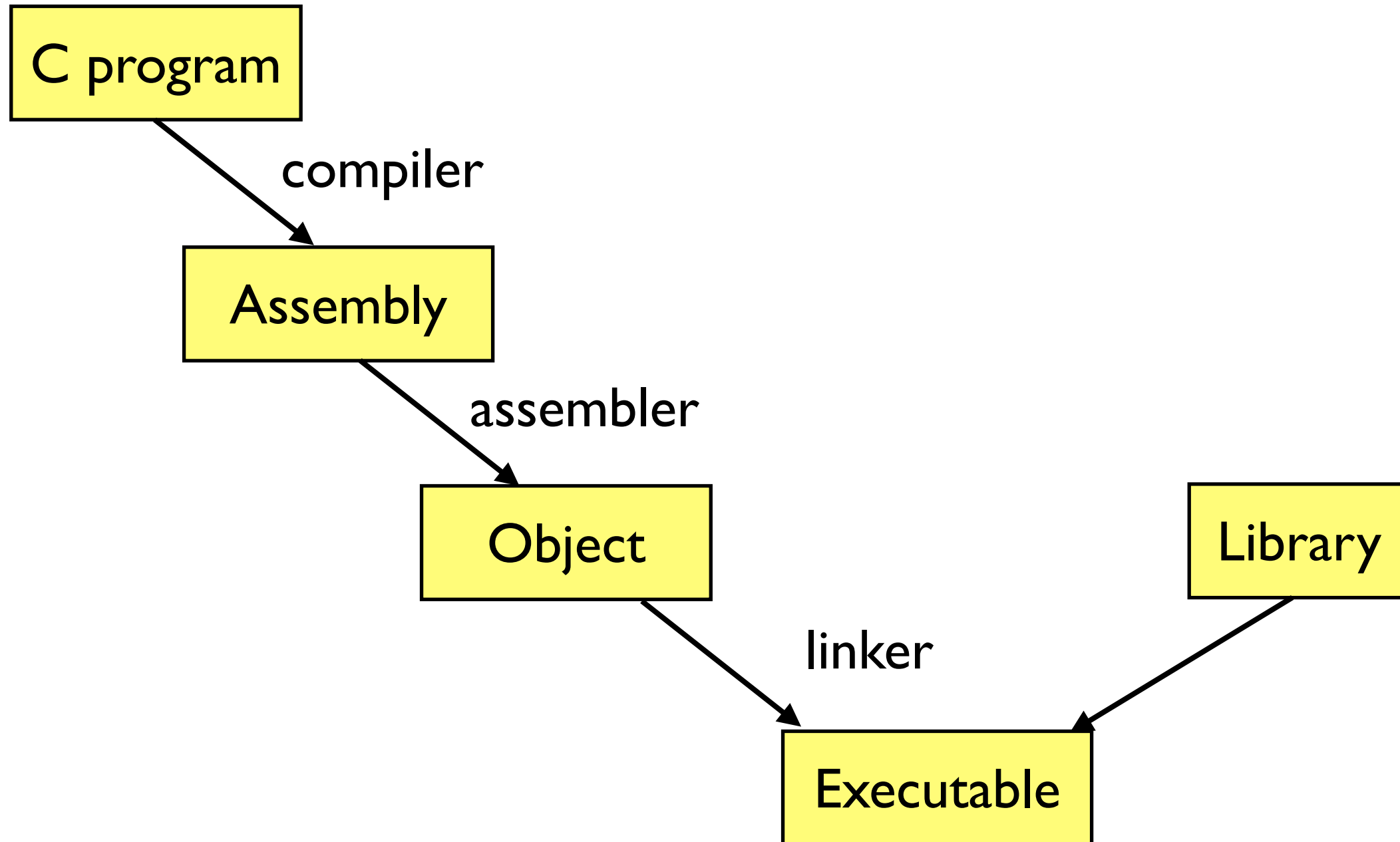
C to Assembly



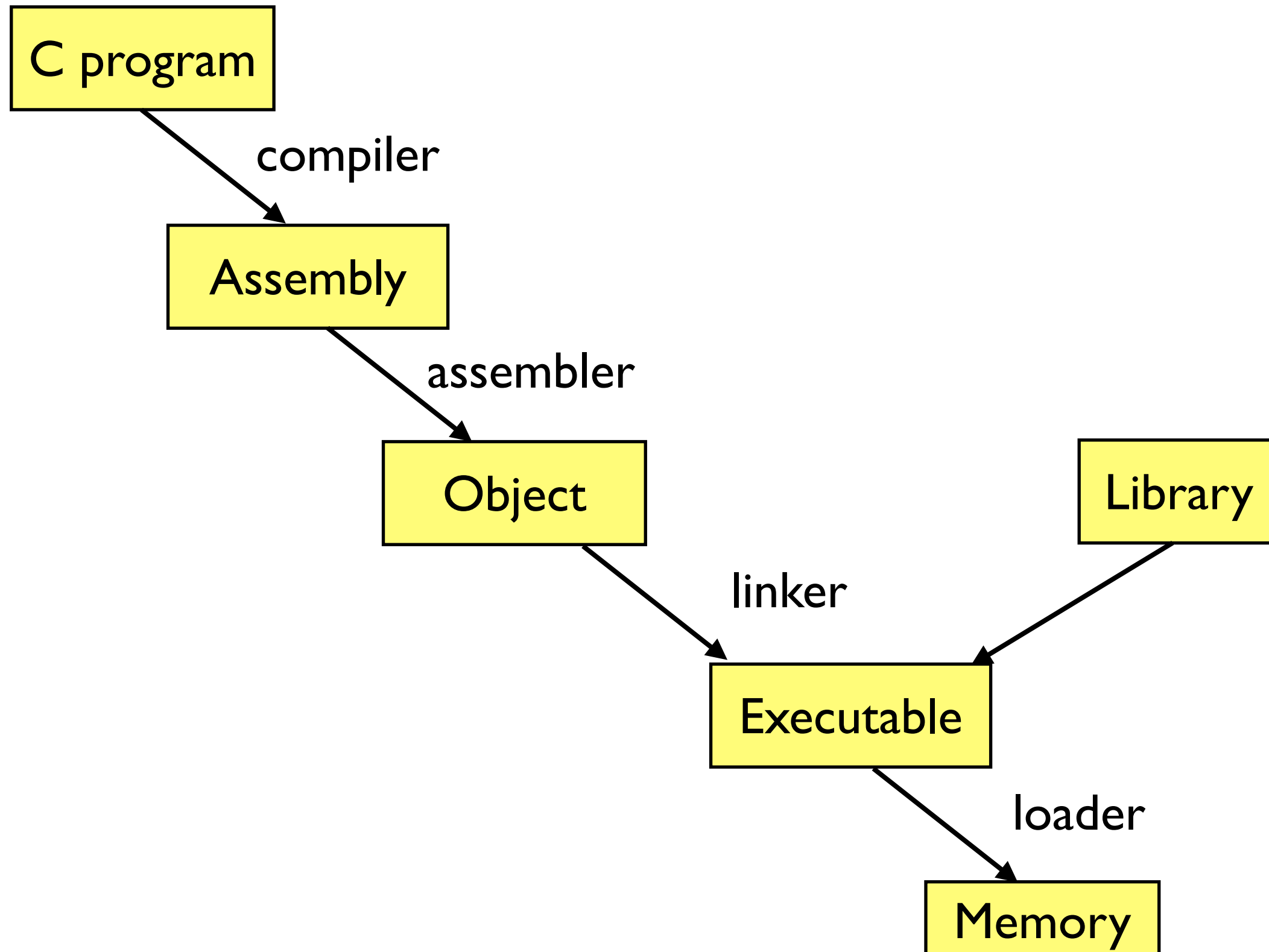
C to Assembly



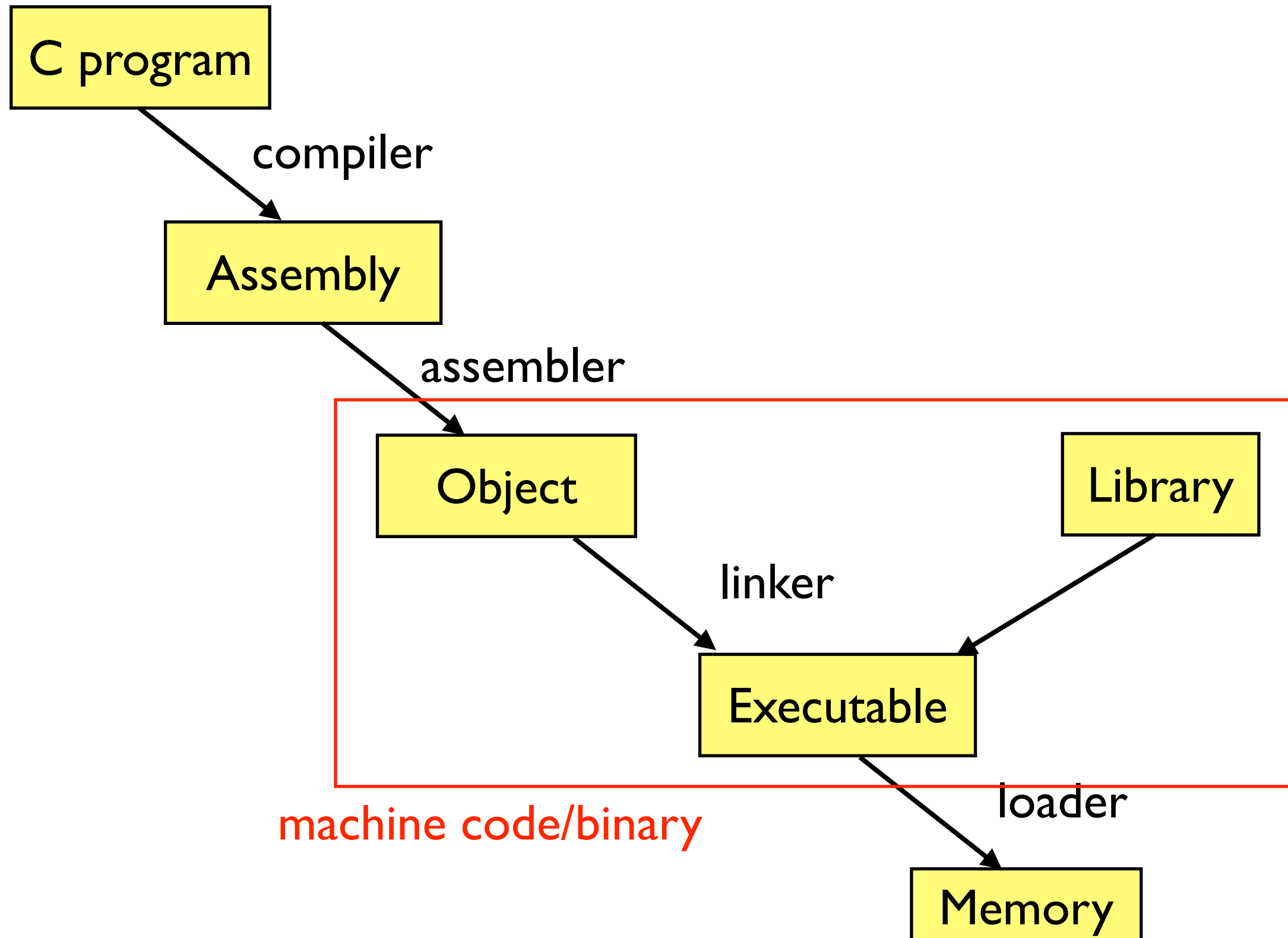
C to Assembly



C to Assembly



C to Assembly



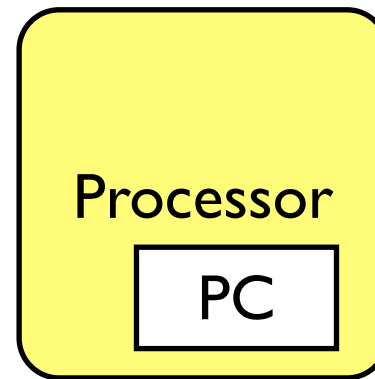
Then What?

Then What?

instruction memory

120007a30:	0f00bb27	ldah	gp,15(t12)
120007a34:	509cbd23	lda	gp,-25520(gp)
120007a38:	00005d24	ldah	t1,0(gp)
120007a3c:	0000bd24	ldah	t4,0(gp)
120007a40:	2ca422a0	ldl	t0,-23508(t1)
120007a44:	130020e4	beq	t0,120007a94
120007a48:	00003d24	ldah	t0,0(gp)
120007a4c:	2ca4e2b3	stl	zero,-23508(t1)
120007a50:	0004ff47	clr	v0
120007a54:	28a4e5b3	stl	zero,-23512(t4)
120007a58:	20a421a4	ldq	t0,-23520(t0)
120007a5c:	0e0020e4	beq	t0,120007a98
120007a60:	0204e147	mov	t0,t1
120007a64:	0304ff47	clr	t2
120007a68:	0500e0c3	br	120007a80

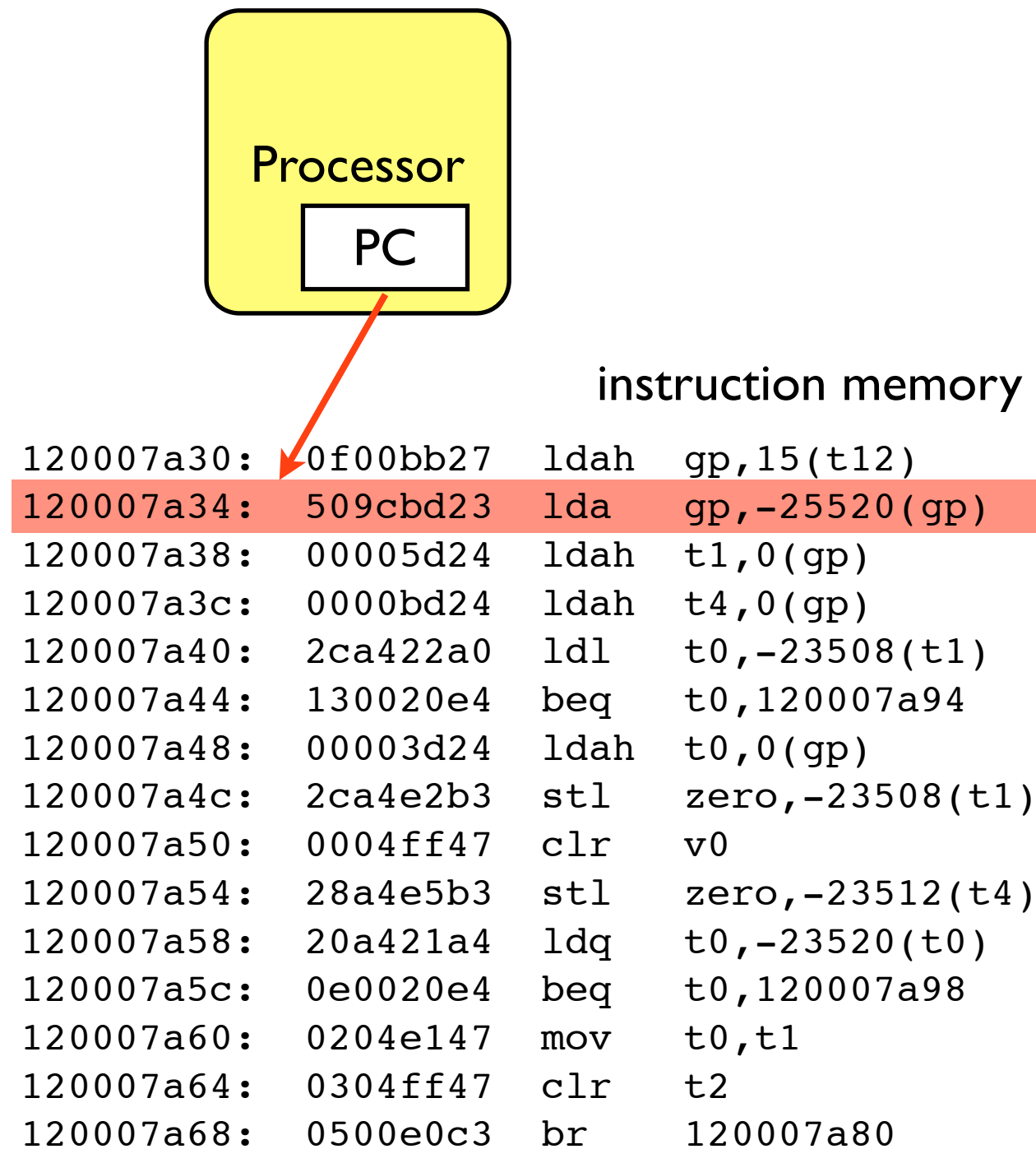
Then What?



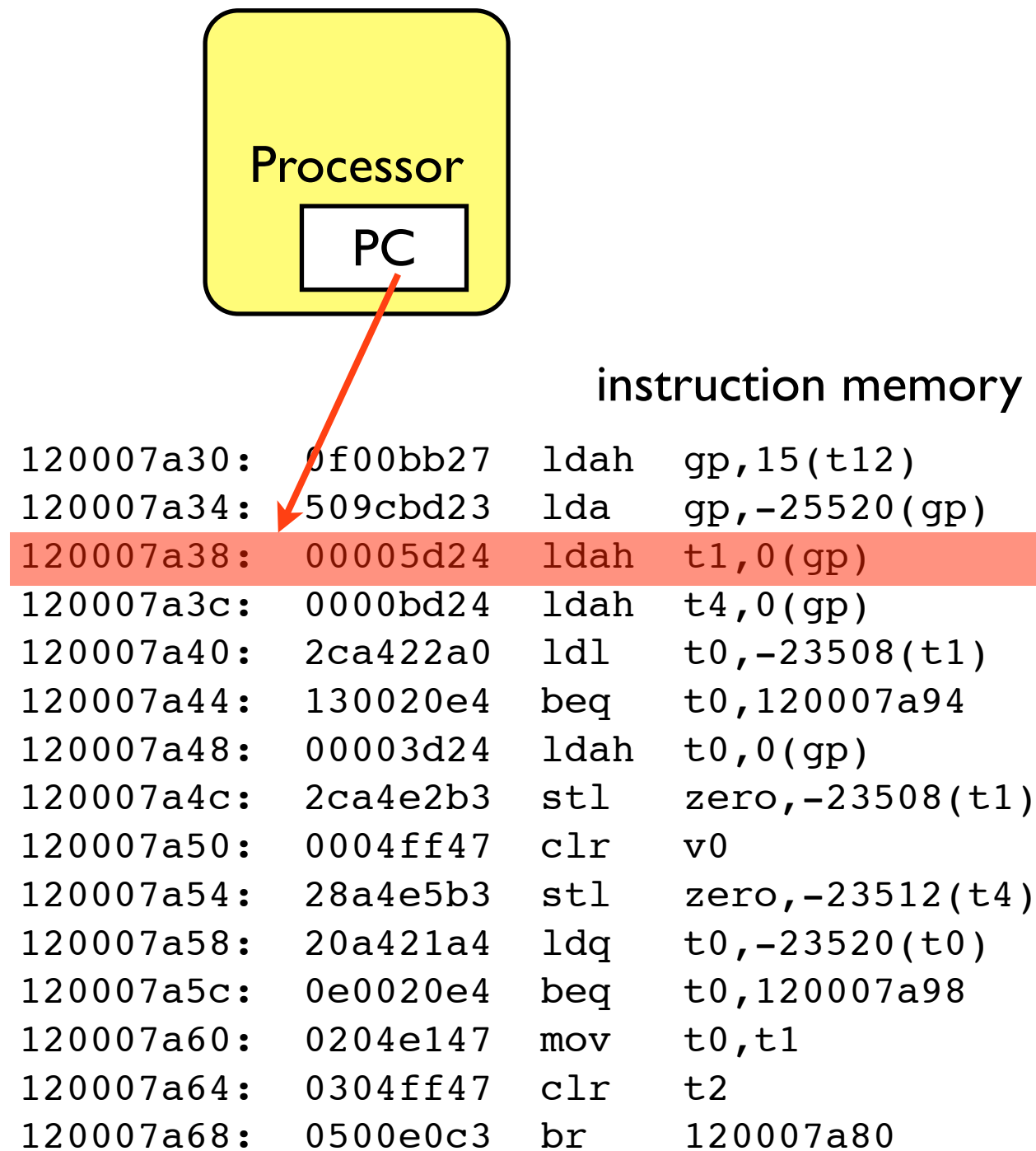
instruction memory

120007a30:	0f00bb27	ldah	gp,15(t12)
120007a34:	509cbd23	lda	gp,-25520(gp)
120007a38:	00005d24	ldah	t1,0(gp)
120007a3c:	0000bd24	ldah	t4,0(gp)
120007a40:	2ca422a0	ldl	t0,-23508(t1)
120007a44:	130020e4	beq	t0,120007a94
120007a48:	00003d24	ldah	t0,0(gp)
120007a4c:	2ca4e2b3	stl	zero,-23508(t1)
120007a50:	0004ff47	clr	v0
120007a54:	28a4e5b3	stl	zero,-23512(t4)
120007a58:	20a421a4	ldq	t0,-23520(t0)
120007a5c:	0e0020e4	beq	t0,120007a98
120007a60:	0204e147	mov	t0,t1
120007a64:	0304ff47	clr	t2
120007a68:	0500e0c3	br	120007a80

Then What?



Then What?



What Must We Put in an Instruction?

- An instruction must drive the Execution Cycle

What Must We Put in an Instruction?

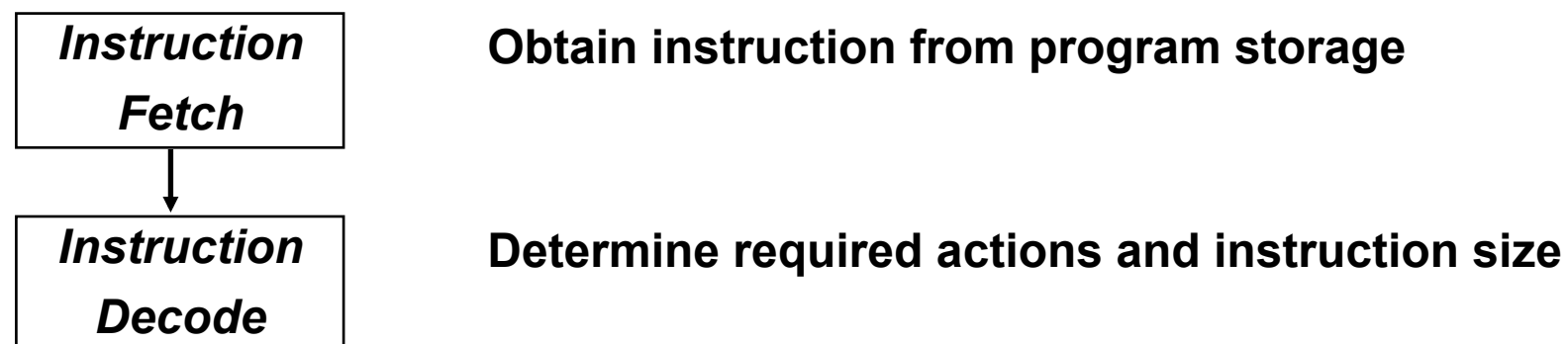
- An instruction must drive the Execution Cycle

***Instruction
Fetch***

Obtain instruction from program storage

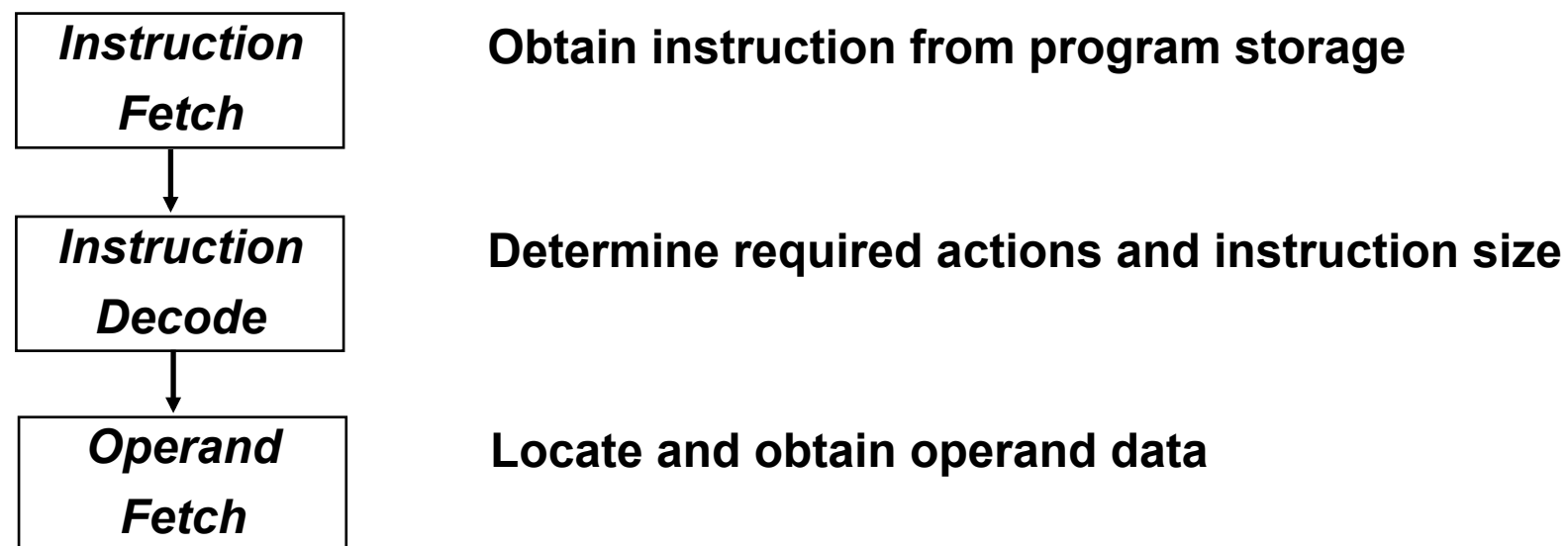
What Must We Put in an Instruction?

- An instruction must drive the Execution Cycle



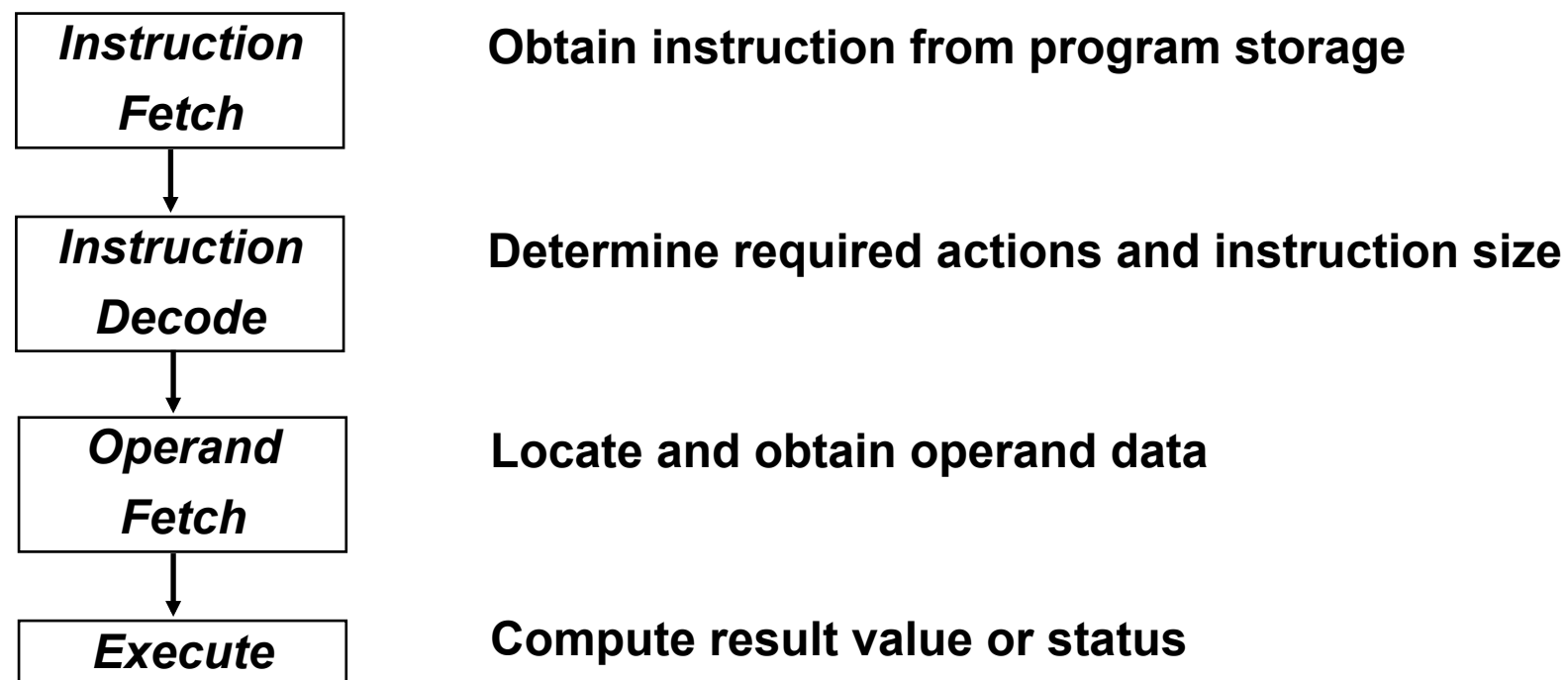
What Must We Put in an Instruction?

- An instruction must drive the Execution Cycle



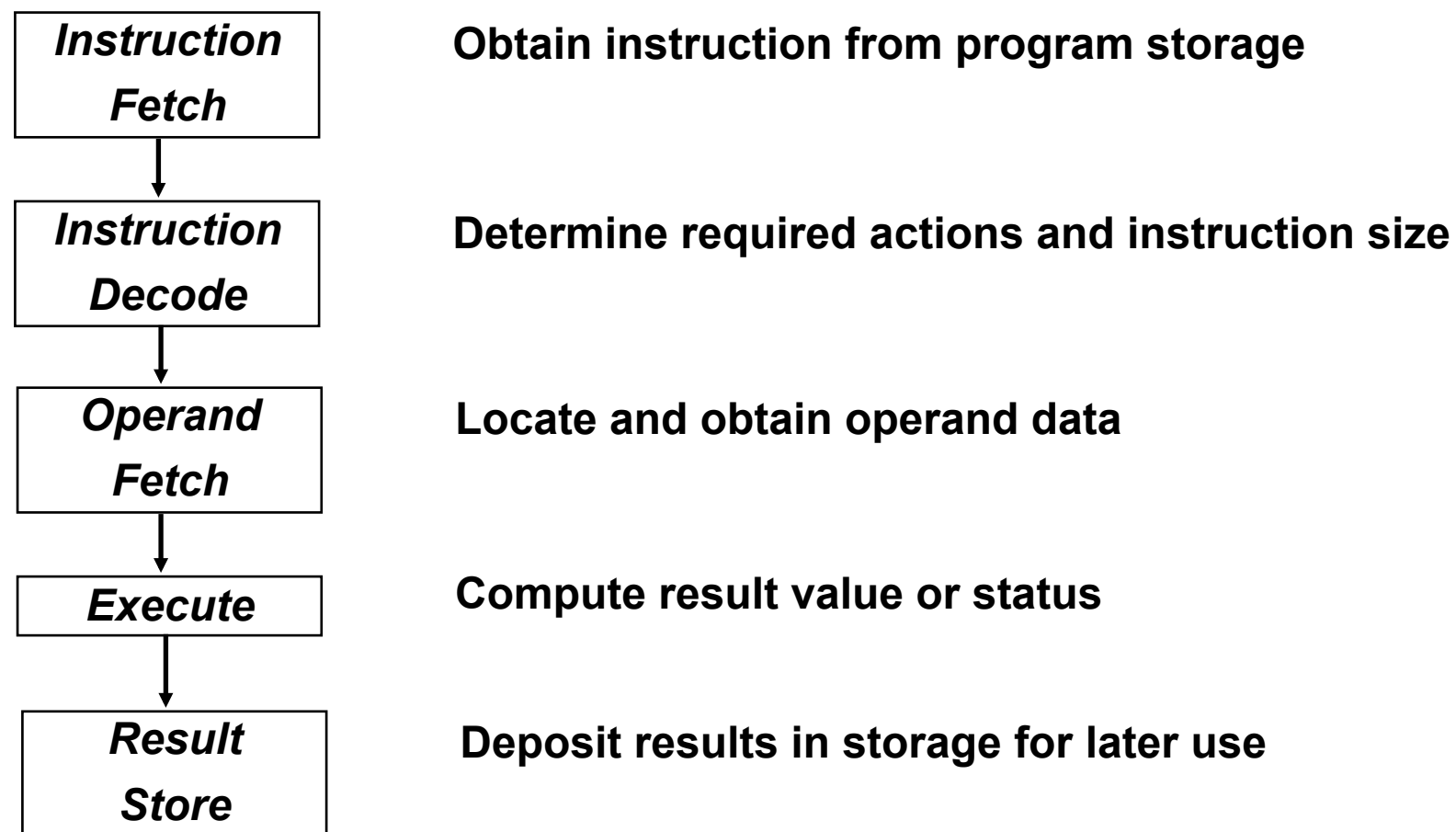
What Must We Put in an Instruction?

- An instruction must drive the Execution Cycle



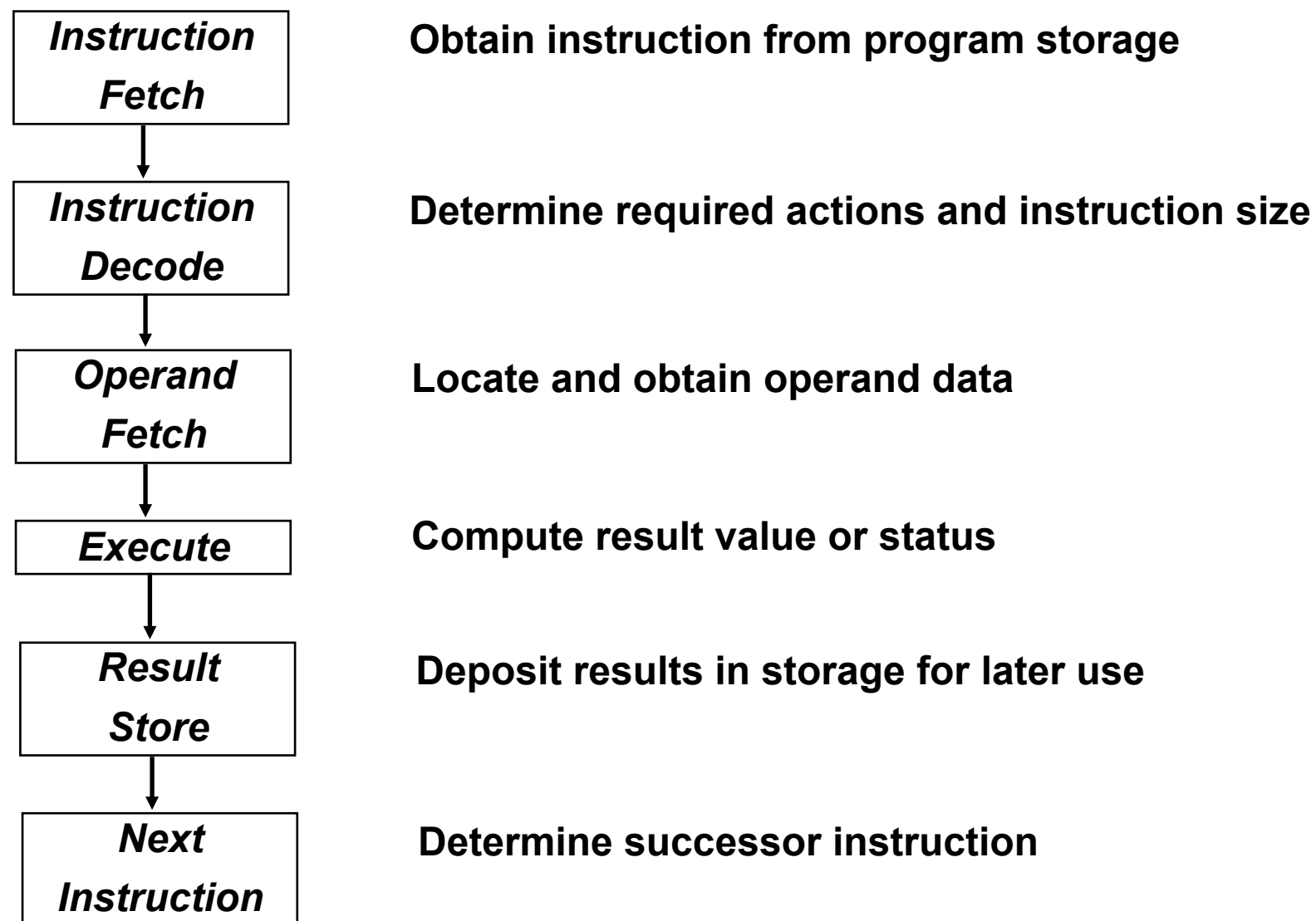
What Must We Put in an Instruction?

- An instruction must drive the Execution Cycle



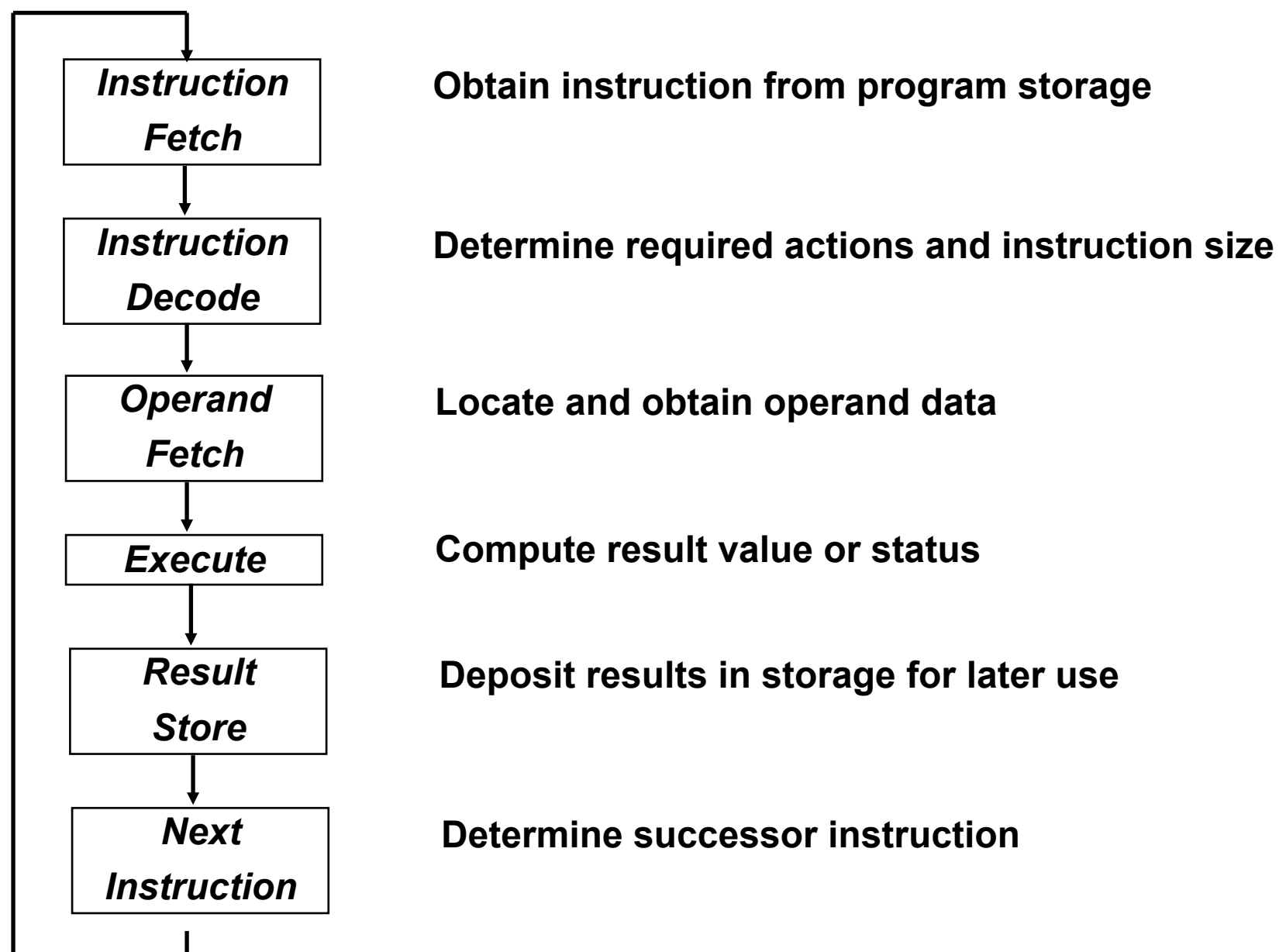
What Must We Put in an Instruction?

- An instruction must drive the Execution Cycle

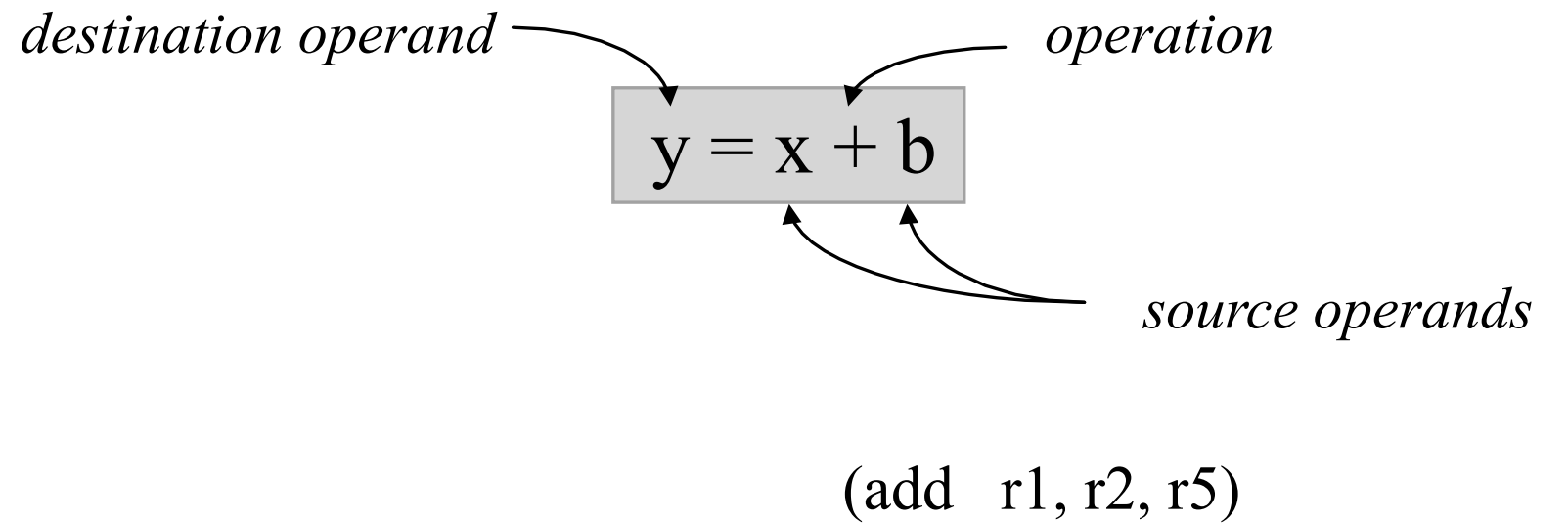


What Must We Put in an Instruction?

- An instruction must drive the Execution Cycle

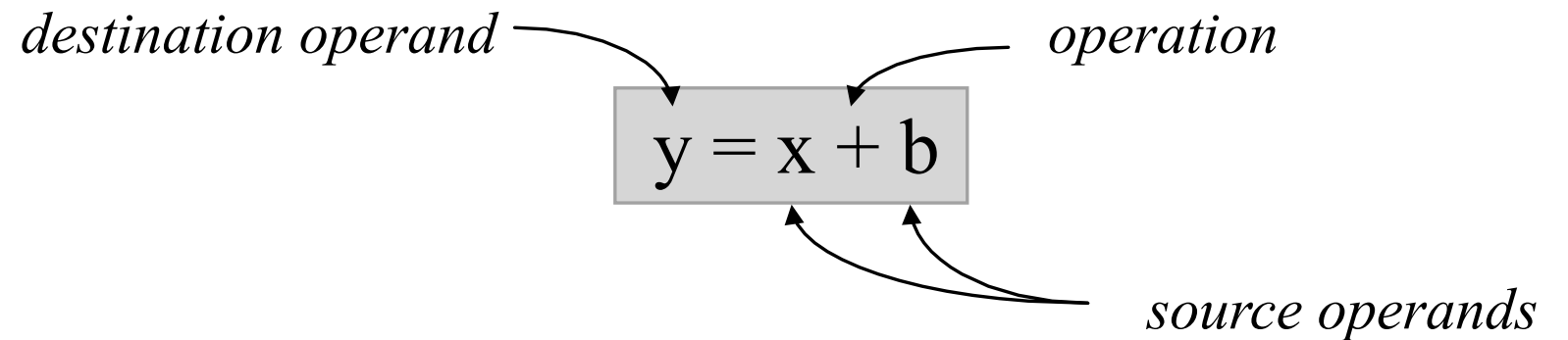


Constructing an ISA: Key Decisions



Constructing an ISA: Key Decisions

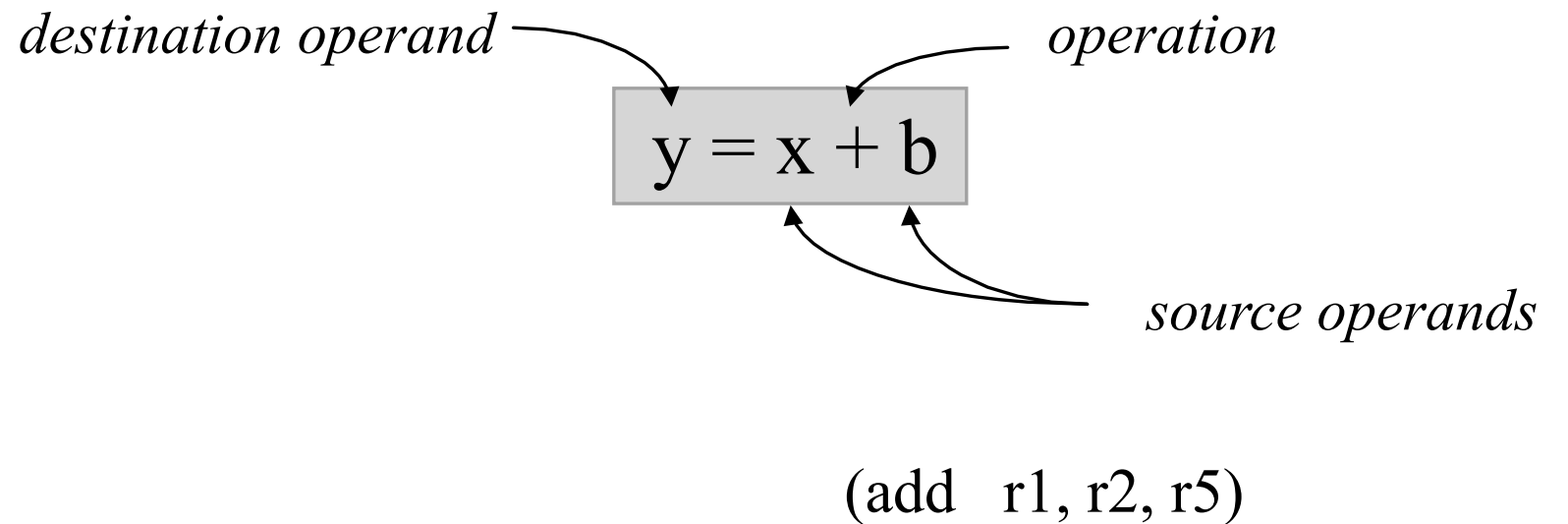
- Operations
 - How many?
 - Which ones



(add r1, r2, r5)

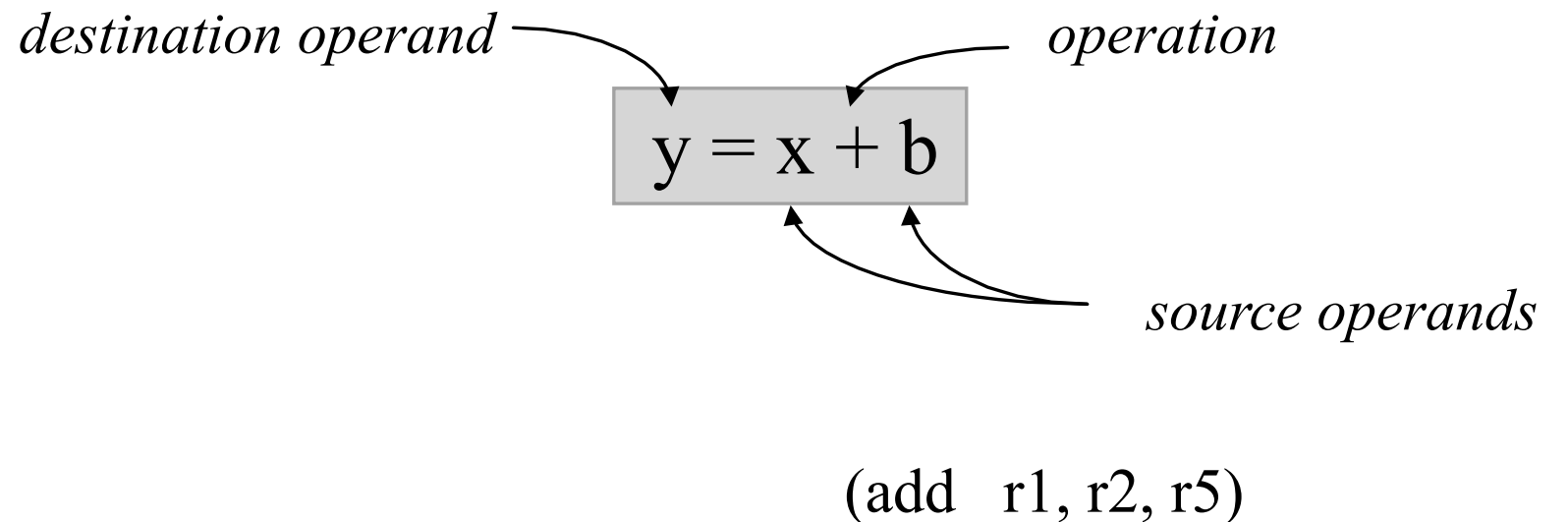
Constructing an ISA: Key Decisions

- Operations
 - How many?
 - Which ones
- Operands
 - How many?
 - Location
 - Types
 - How to specify?



Constructing an ISA: Key Decisions

- Operations
 - How many?
 - Which ones
- Operands
 - How many?
 - Location
 - Types
 - How to specify?
- Instruction format
 - Size
 - How many formats?



*how does the computer know what
0001 0100 1101 1111
means?*

Crafting an ISA

- We'll look at some of the decisions facing an instruction set architect, and
- how those decisions were made in the design of the MIPS instruction set.
- (Will be important as you create your own ISA ;-))



Instruction Length

Instruction Length

Fixed Length

32 bits

opcode	rs	rt	rd	shift amount	funct
--------	----	----	----	-----------------	-------

32 bits

opcode	rs	rt	immediate / offset
--------	----	----	--------------------

32 bits

opcode	target
--------	--------

32 bits

opcode	rs	rt	rd	shift amount	funct
--------	----	----	----	-----------------	-------

32 bits

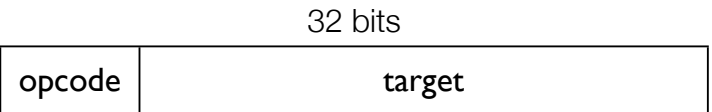
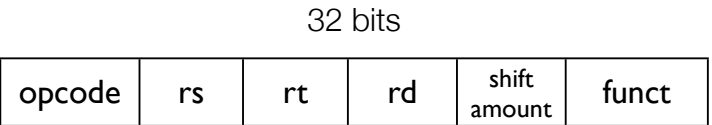
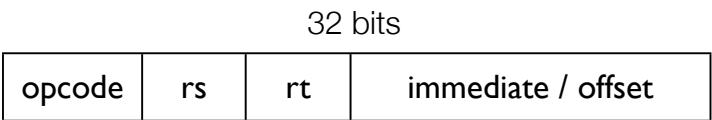
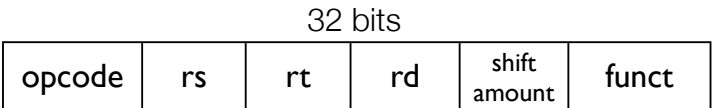
opcode	target
--------	--------

32 bits

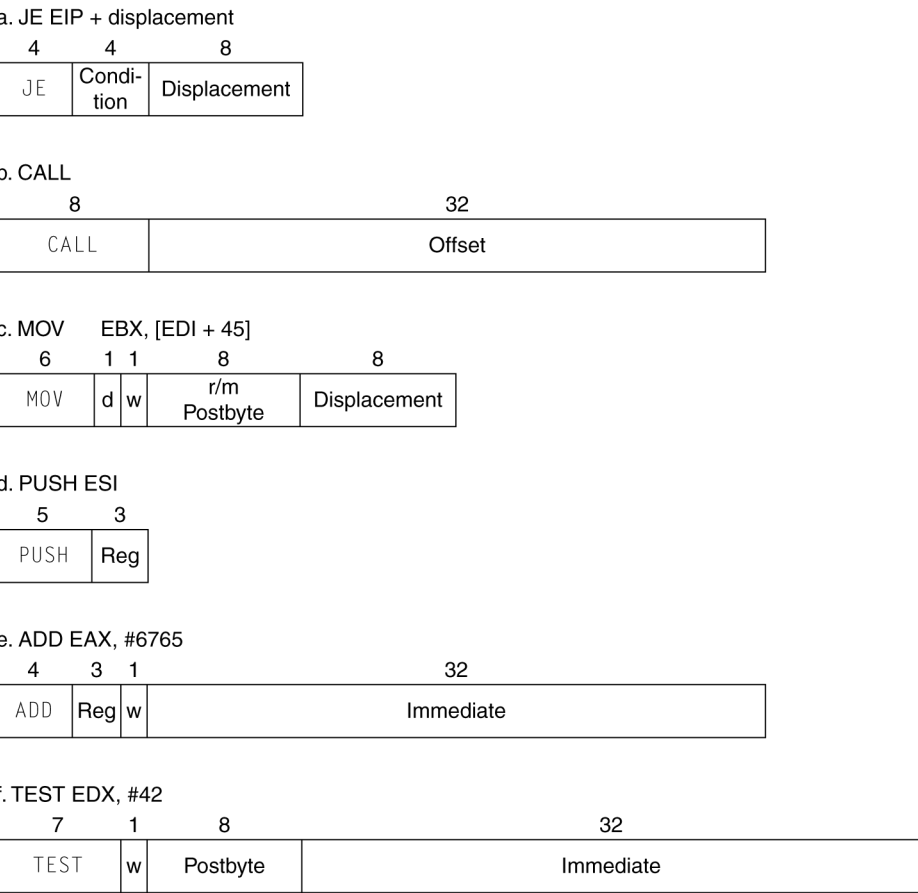
opcode	rs	rt	immediate / offset
--------	----	----	--------------------

Instruction Length

Fixed Length



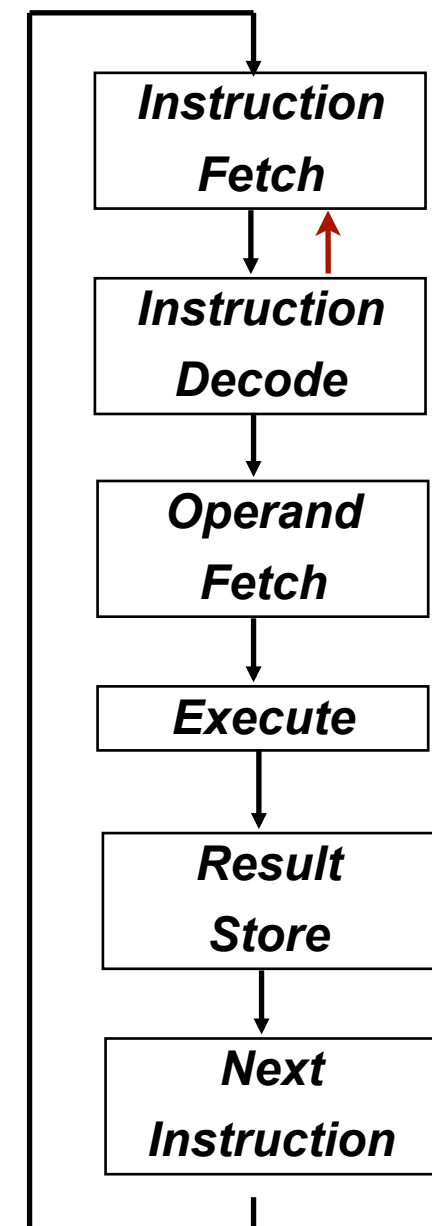
Variable Length



Instruction Length

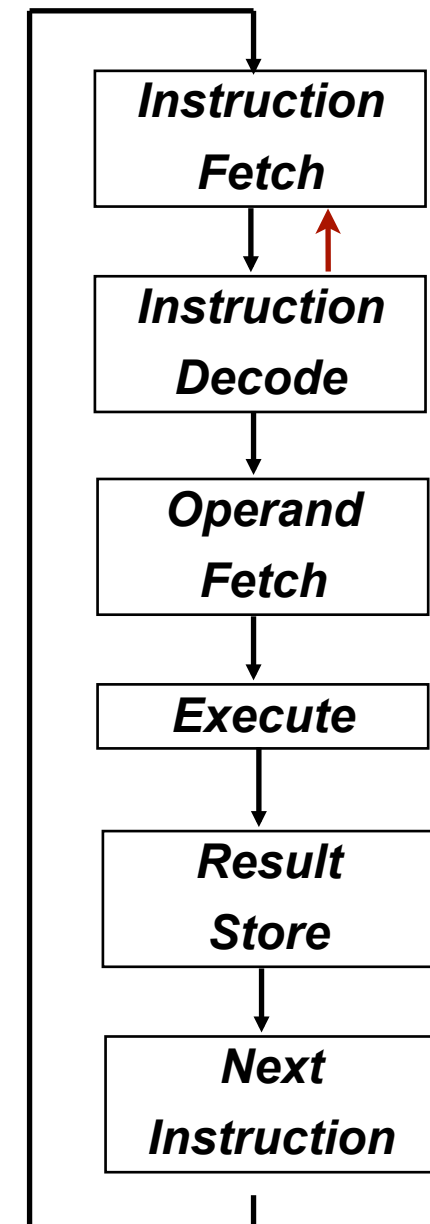
Instruction Length

- Variable-length instructions (Intel 80x86, VAX) require multi-step fetch and decode, but allow for a much more flexible and compact instruction set.



Instruction Length

- Variable-length instructions (Intel 80x86, VAX) require multi-step fetch and decode, but allow for a much more flexible and compact instruction set.
- Fixed-length instructions allow easy fetch and decode, and simplify pipelining and parallelism.
- All MIPS instructions are 32 bits long.
 - this decision impacts every other ISA decision we make because it makes instruction bits scarce.



Instruction Formats

Instruction Formats

- What does each bit mean?

Instruction Formats

- What does each bit mean?
- Having many different instruction formats...

Instruction Formats

- What does each bit mean?
- Having many different instruction formats...
 - complicates decoding

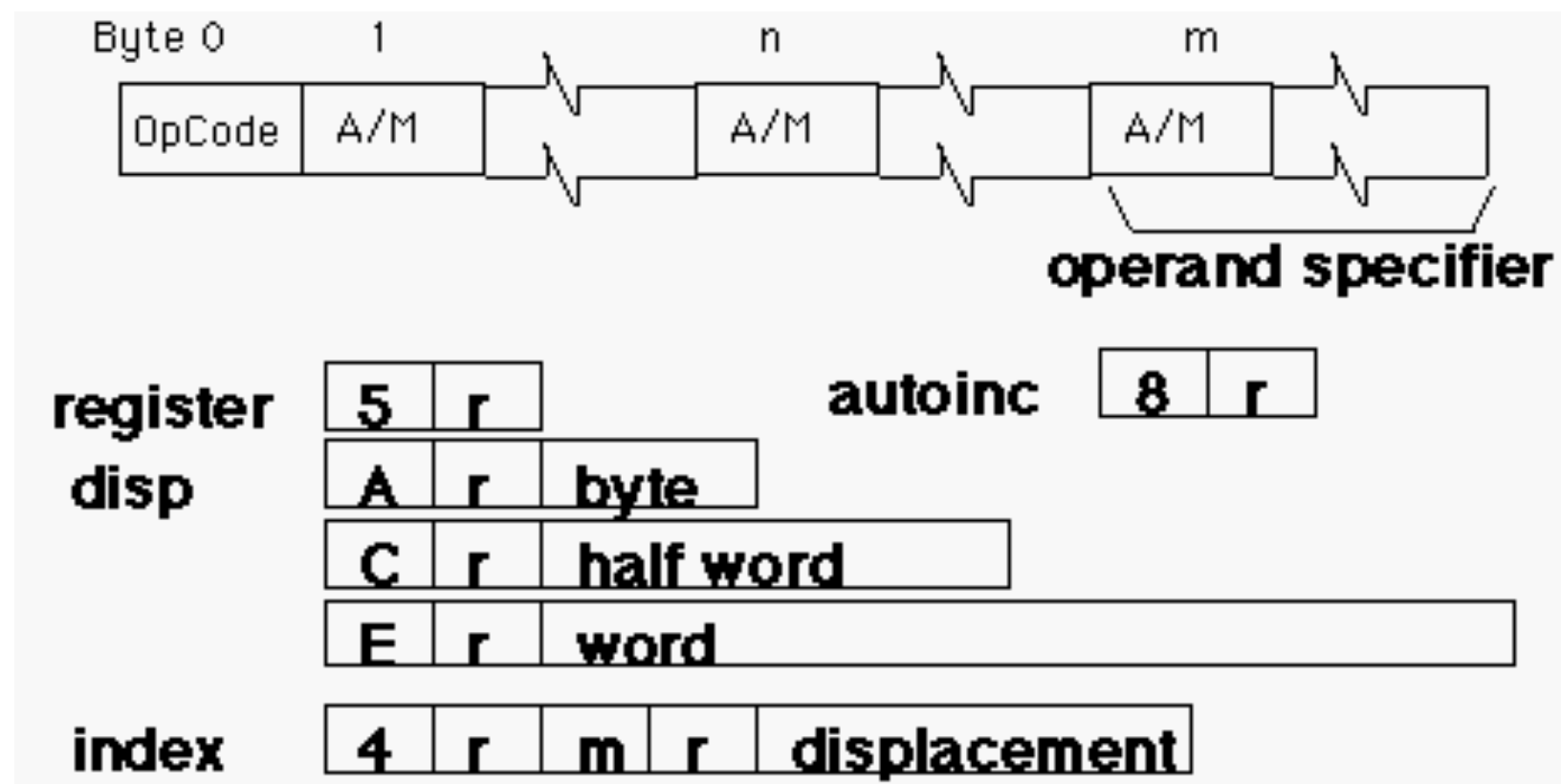
Instruction Formats

- What does each bit mean?
- Having many different instruction formats...
 - complicates decoding
 - uses more instruction bits (to specify the format)

Instruction Formats

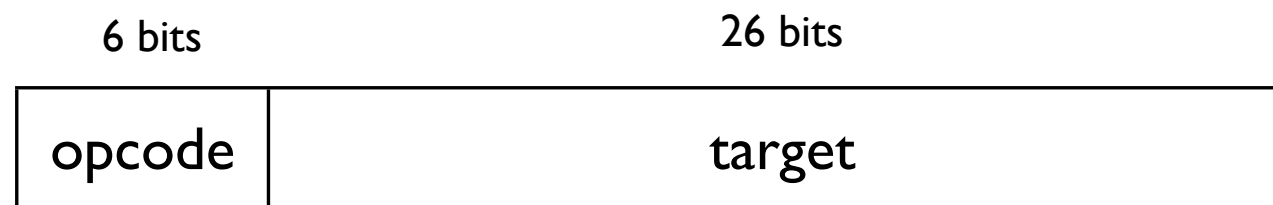
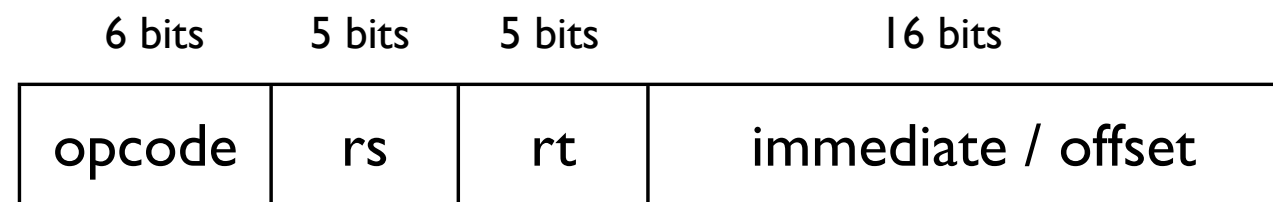
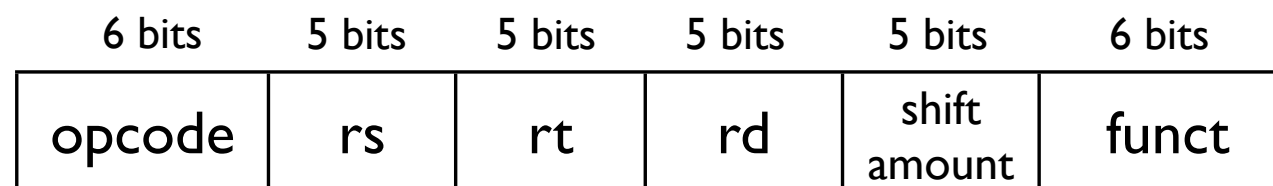
- What does each bit mean?
- Having many different instruction formats...
 - complicates decoding
 - uses more instruction bits (to specify the format)

VAX 11 instruction format



What did MIPS do?

- MIPS has 3 formats



- The opcode tells the machine which format
 - so `add r1, r2, r3` has
 - `opcode=0, funct=32, rs=2, rt=3, rd=1, sa=0`
 - `000000 00010 00011 00001 00000 100000`

Accessing the Operands

- Operands are generally in one of two places:
 - Registers (32 int, 32 fp)
 - Memory (2^{32} locations)
- Registers are
 - Easy to specify
 - Close to the processor (fast access)
- The idea that we want to access registers whenever possible led to load-store architectures. (vs register-memory e.g., x86)
 - Normal arithmetic instructions only access registers
 - Only access memory with explicit loads and stores

Load-store Architectures

- can do:
 - add $r1 = r2 + r3$
 - and
 - load $r3, M(\text{address})$
 - Forces heavy dependence on registers, which is exactly what you want in today's CPUs
- can't do
 - add $r1 = r2 + M(\text{address})$
 - - More instructions
 - + Fast implementation (e.g., easy pipelining)
 - *Elegant*

Load-store Architectures

- can do:
 - add $r1 = r2 + r3$
 - and
 - load $r3, M(\text{address})$
 - Forces heavy dependence on registers, which is exactly what you want in today's CPUs
- can't do
 - add $r1 = r2 + M(\text{address})$
 - - More instructions
 - + Fast implementation (e.g., easy pipelining)
 - *Elegant*

Why else?

How Many Operands?

- Most instructions have three operands (e.g., $z = x + y$).
- Well-known ISAs specify 0-3 (explicit) operands per instruction.
- Operands can be specified implicitly or explicitly.

How Many Operands - ISA Classes

How Many Operands - ISA Classes

Accumulator:

1 address

add A

$\text{acc} \leftarrow \text{acc} + \text{mem}[A]$

How Many Operands - ISA Classes

Accumulator:

1 address

add A

$\text{acc} \leftarrow \text{acc} + \text{mem}[A]$

Stack:

0 address

add

$\text{tos} \leftarrow \text{tos} + \text{next}$

How Many Operands - ISA Classes

Accumulator:

1 address

add A

$\text{acc} \leftarrow \text{acc} + \text{mem}[A]$

Stack:

0 address

add

$\text{tos} \leftarrow \text{tos} + \text{next}$

General Purpose Register:

2 address

add A B

$\text{EA}(A) \leftarrow \text{EA}(A) + \text{EA}(B)$

3 address

add A B C

$\text{EA}(A) \leftarrow \text{EA}(B) + \text{EA}(C)$

How Many Operands - ISA Classes

Accumulator:

1 address

add A

$\text{acc} \leftarrow \text{acc} + \text{mem}[A]$

Stack:

0 address

add

$\text{tos} \leftarrow \text{tos} + \text{next}$

General Purpose Register:

2 address

add A B

$\text{EA}(A) \leftarrow \text{EA}(A) + \text{EA}(B)$

3 address

add A B C

$\text{EA}(A) \leftarrow \text{EA}(B) + \text{EA}(C)$

Load/Store:

3 address

add Ra Rb Rc

$\text{Ra} \leftarrow \text{Rb} + \text{Rc}$

load Ra Rb

$\text{Ra} \leftarrow \text{mem}[\text{Rb}]$

store Ra Rb

$\text{mem}[\text{Rb}] \leftarrow \text{Ra}$

How Many Operands - ISA Classes

Code sequence for $C = A + B$ for four classes of instruction sets:

Stack

Accumulator

GP Register

GP Register

(register-memory)

(load-store)

How Many Operands - ISA Classes

Code sequence for $C = A + B$ for four classes of instruction sets:

Stack

Accumulator

GP Register

GP Register

(register-memory)

(load-store)

Push A

Push B

Add

Pop C

How Many Operands - ISA Classes

Code sequence for $C = A + B$ for four classes of instruction sets:

Stack

Accumulator

GP Register

GP Register

(register-memory)

(load-store)

Push A

Load A

Push B

Add B

Add

Store C

Pop C

How Many Operands - ISA Classes

Code sequence for $C = A + B$ for four classes of instruction sets:

<u>Stack</u>	<u>Accumulator</u>	<u>GP Register</u> (register-memory)	<u>GP Register</u> (load-store)
Push A	Load A	ADD C, A, B	
Push B	Add B		
Add	Store C		
Pop C			

How Many Operands - ISA Classes

Code sequence for $C = A + B$ for four classes of instruction sets:

<u>Stack</u>	<u>Accumulator</u>	<u>GP Register</u> (register-memory)	<u>GP Register</u> (load-store)
Push A	Load A	ADD C, A, B	Load R1,A
Push B	Add B		Load R2,B
Add	Store C		Add R3,R1,R2
Pop C			Store C,R3

Addressing Modes

Addressing Modes

- **Register direct R3**

Addressing Modes

- **Register direct** **R3**
- **Immediate (literal)** **#25**

Addressing Modes

- **Register direct** R3
- **Immediate (literal)** #25
- **Direct (absolute)** M[10000]

Addressing Modes

- **Register direct** **R3**
- **Immediate (literal)** **#25**
- **Direct (absolute)** **M[10000]**

- **Register indirect** **M[R3]**

Addressing Modes

- **Register direct** $R3$
- **Immediate (literal)** $\#25$
- **Direct (absolute)** $M[10000]$

- **Register indirect** $M[R3]$
- **Base+Displacement** $M[R3 + 10000]$

Addressing Modes

- **Register direct** $R3$
- **Immediate (literal)** $\#25$
- **Direct (absolute)** $M[10000]$

- **Register indirect** $M[R3]$
- **Base+Displacement** $M[R3 + 10000]$
- **Base+Index** $M[R3 + R4]$

Addressing Modes

- Register direct $R3$
- Immediate (literal) $\#25$
- Direct (absolute) $M[10000]$

- Register indirect $M[R3]$
- Base+Displacement $M[R3 + 10000]$
- Base+Index $M[R3 + R4]$
- Scaled Index $M[R3 + R4 * d + 10000]$

Addressing Modes

- **Register direct** $R3$
- **Immediate (literal)** $\#25$
- **Direct (absolute)** $M[10000]$

- **Register indirect** $M[R3]$
- **Base+Displacement** $M[R3 + 10000]$
- **Base+Index** $M[R3 + R4]$
- **Scaled Index** $M[R3 + R4 * d + 10000]$
- **Autoincrement** $M[R3++]$

Addressing Modes

- **Register direct** $R3$
- **Immediate (literal)** $\#25$
- **Direct (absolute)** $M[10000]$

- **Register indirect** $M[R3]$
- **Base+Displacement** $M[R3 + 10000]$
- **Base+Index** $M[R3 + R4]$
- **Scaled Index** $M[R3 + R4 * d + 10000]$
- **Autoincrement** $M[R3++]$
- **Autodecrement** $M[R3--]$

Addressing Modes

- Register direct $R3$
- Immediate (literal) $\#25$
- Direct (absolute) $M[10000]$
- Register indirect $M[R3]$
- Base+Displacement $M[R3 + 10000]$
- Base+Index $M[R3 + R4]$
- Scaled Index $M[R3 + R4 * d + 10000]$
- Autoincrement $M[R3++]$
- Autodecrement $M[R3--]$
- Memory Indirect $M[M[R3]]$

What did MIPS do?

register direct

OP	rs	rt	rd	sa	funct
----	----	----	----	----	-------

add \$1, \$2, \$3

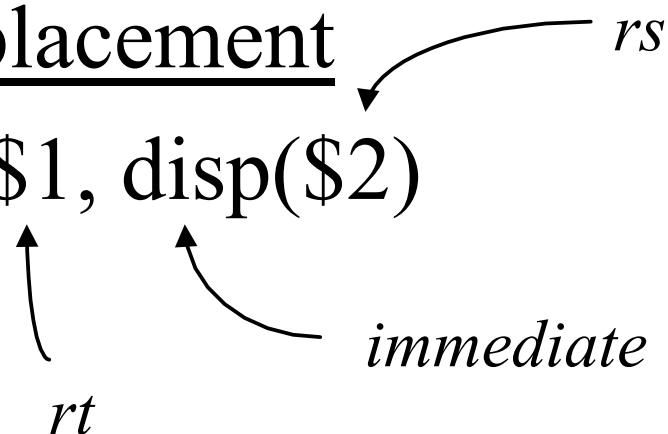
immediate

OP	rs	rt	immediate
----	----	----	-----------

add \$1, \$2, #35

base + displacement

lw \$1, disp(\$2)



$$(R1 = M[R2 + disp])$$

What did MIPS do?

register direct

OP	rs	rt	rd	sa	funct
----	----	----	----	----	-------

add \$1, \$2, \$3

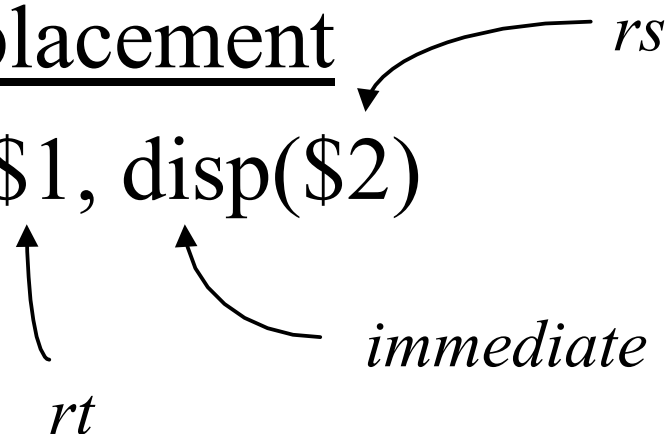
immediate

OP	rs	rt	immediate
----	----	----	-----------

add \$1, \$2, #35

base + displacement

lw \$1, disp(\$2)



$(R1 = M[R2 + disp])$

We get register indirect and absolute for free, but how?

What did MIPS do?

register direct

OP	rs	rt	rd	sa	funct
----	----	----	----	----	-------

add \$1, \$2, \$3

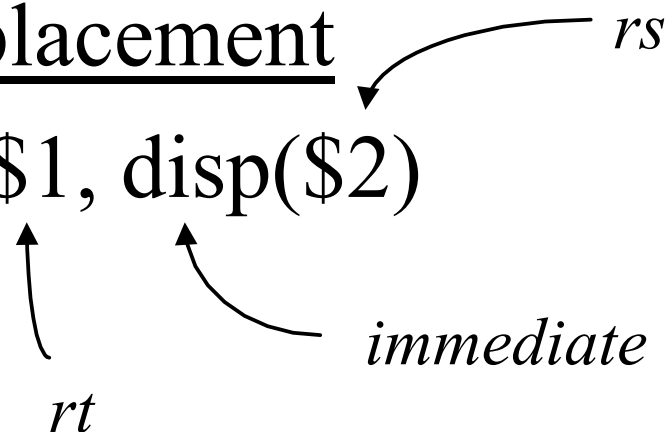
immediate

OP	rs	rt	immediate
----	----	----	-----------

add \$1, \$2, #35

base + displacement

lw \$1, disp(\$2)



$(R1 = M[R2 + disp])$

We get register indirect and absolute for free, but how?

register indirect

$\Rightarrow disp = 0$

absolute

$\Rightarrow (rs) = 0$

In Previous Class...

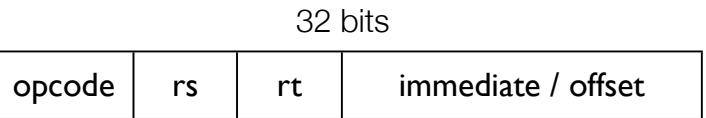
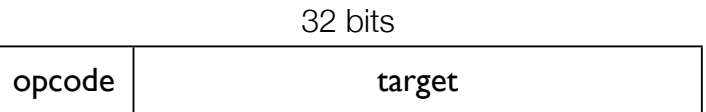
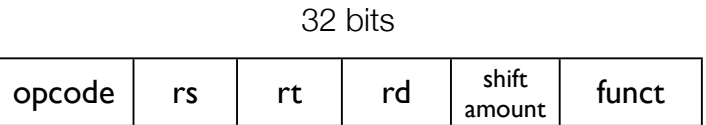
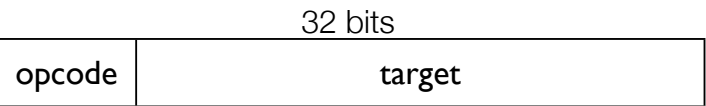
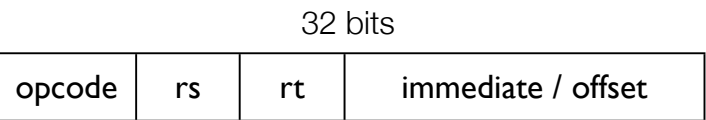
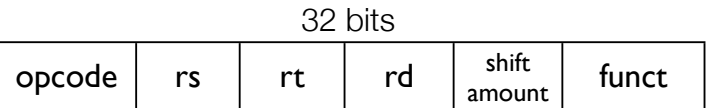
Last Class

- ISA Design Decisions
 - Instruction Length
 - Instruction Formats
 - Addressing Modes
- What Did MIPS Do?

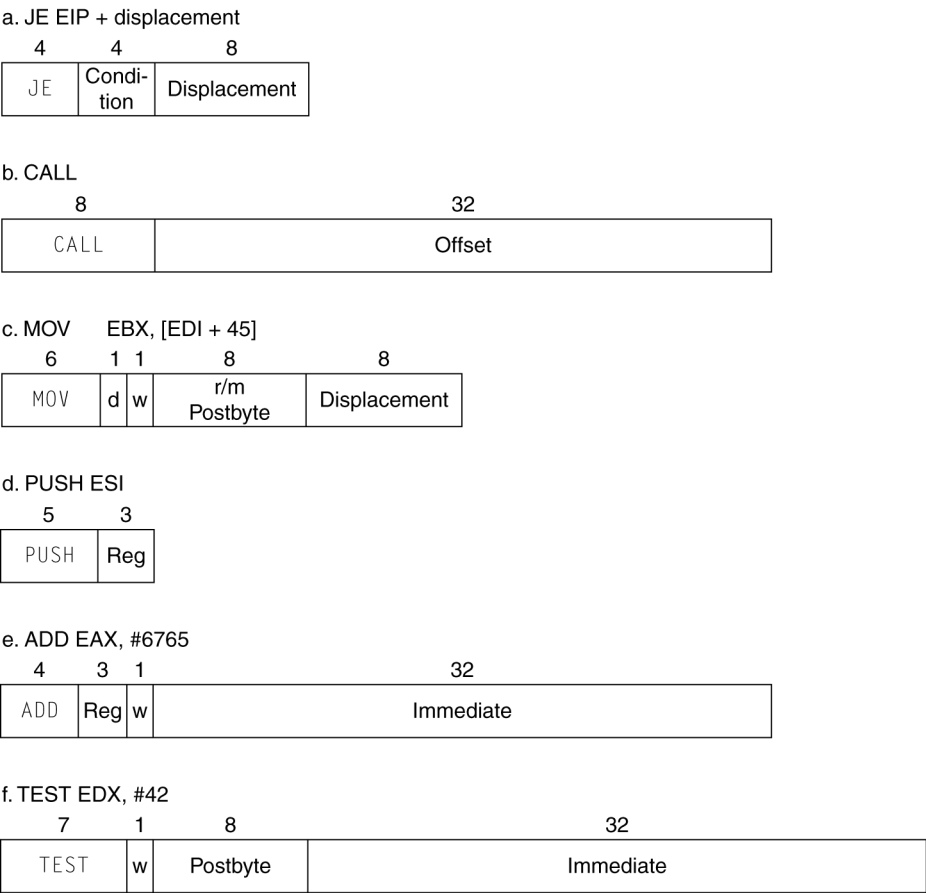


Instruction Length

Fixed Length

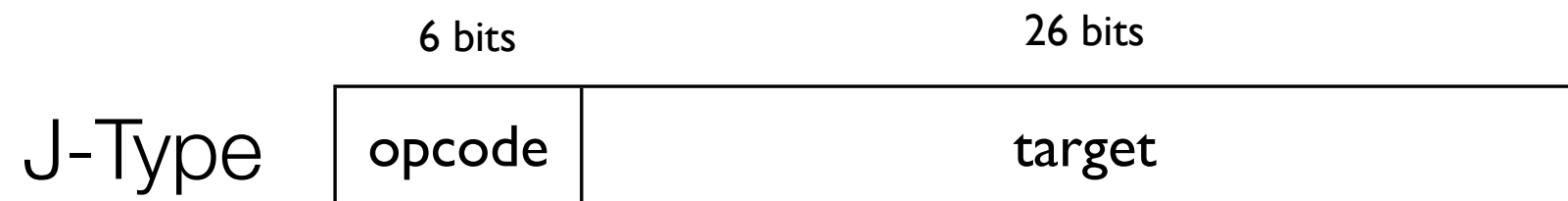
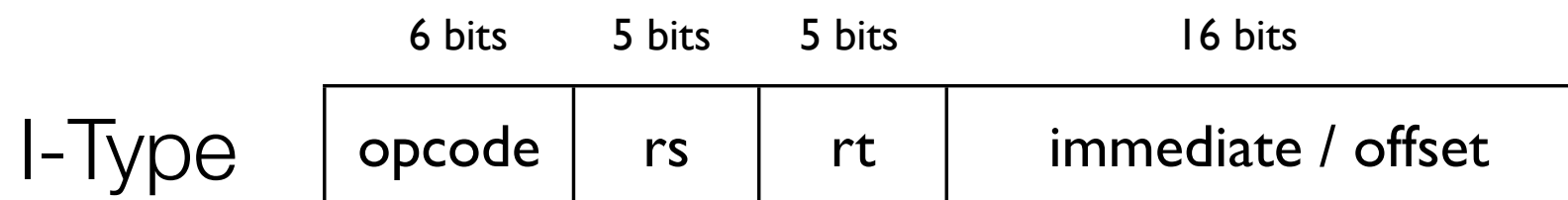
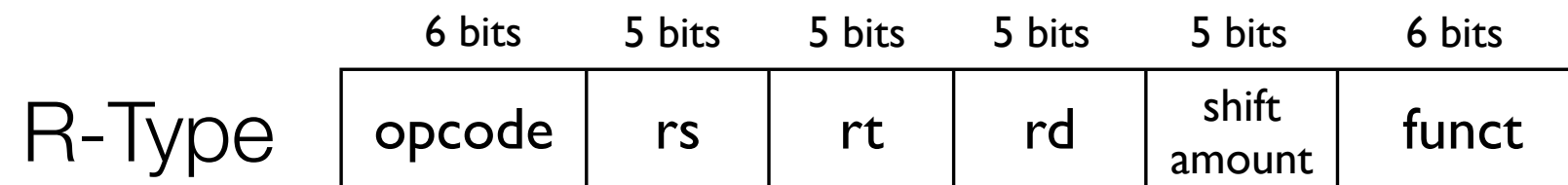


Variable Length



Instruction Format (What did MIPS do?)

- MIPS has 3 formats



- The opcode tells the machine which format
 - so `add r1, r2, r3` has
 - `opcode=0, funct=32, rs=2, rt=3, rd=1, sa=0`
 - `000000 00010 00011 00001 00000 100000`

Addressing Modes (What did MIPS do?)

register direct

OP	rs	rt	rd	sa	funct
----	----	----	----	----	-------

add \$1, \$2, \$3

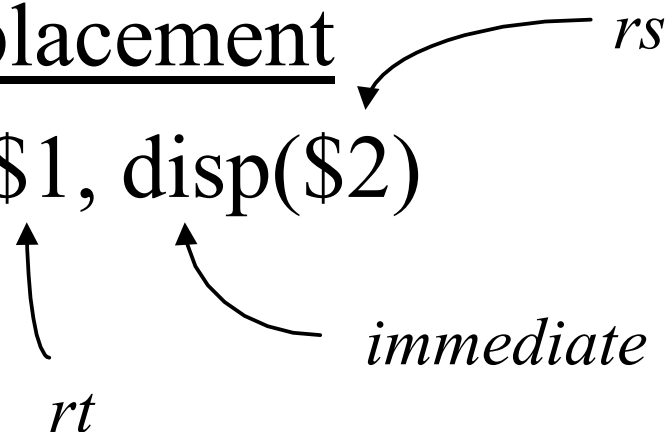
immediate

OP	rs	rt	immediate
----	----	----	-----------

add \$1, \$2, #35

base + displacement

lw \$1, disp(\$2)



$(R1 = M[R2 + disp])$

register indirect

$\Rightarrow disp = 0$

absolute

$\Rightarrow (rs) = 0$

Is It Sufficient

- Is MIPS good enough?
 - Limited addressing modes
 - 16 Bit limit on Immediate
 - Also affect memory operations
- Measurements on the VAX show that these addressing modes (immediate, direct, register indirect, and base+displacement) represent 88% of all addressing mode usage.
- Similar measurements show that 16 bits is enough for the immediate 75 to 80% of the time
- And that 16 bits is enough of a displacement 99% of the time.

btw... Memory Organization

- Viewed as a large, single-dimension array, with an address.
- A memory address is an index into the array
- "Byte addressing" means that the index points to a byte of memory.

0	8 bits of data
1	8 bits of data
2	8 bits of data
3	8 bits of data
4	8 bits of data
5	8 bits of data
6	8 bits of data

...

Memory Organization

- Bytes are nice, but most data items use larger "words"
- For MIPS, a word is 32 bits or 4 bytes.

0	32 bits of data
4	32 bits of data
8	32 bits of data
12	32 bits of data

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
- i.e., what are the least 2 significant bits of a word address?

MIPS ISA thus far...

- Fixed 32-bit instructions
- 3 instruction formats
- 3-operand, load-store architecture
- 32 general-purpose registers (integer, floating point)
 - R0 always equals 0.
- 2 special-purpose integer registers, HI and LO, because multiply and divide produce more than 32 bits.
- Registers are 32-bits wide (word)
- Register, immediate, and base+displacement addressing modes

Whats Left?

- Which instructions?
- Odds and ends

Which Instructions?

Which Instructions?

- Arithmetic

Which Instructions?

- Arithmetic
 - add, subtract, multiply, divide

Which Instructions?

- Arithmetic
 - add, subtract, multiply, divide
- Logical

Which Instructions?

- Arithmetic
 - add, subtract, multiply, divide
- Logical
 - and, or, shift left, shift right

Which Instructions?

- Arithmetic
 - add, subtract, multiply, divide
- Logical
 - and, or, shift left, shift right
- Data transfer

Which Instructions?

- Arithmetic
 - add, subtract, multiply, divide
- Logical
 - and, or, shift left, shift right
- Data transfer
 - load word, store word

Which Instructions?

- Arithmetic
 - add, subtract, multiply, divide
- Logical
 - and, or, shift left, shift right
- Data transfer
 - load word, store word
- Conditional branch

Which Instructions?

- Arithmetic
 - add, subtract, multiply, divide
- Logical
 - and, or, shift left, shift right
- Data transfer
 - load word, store word
- Conditional branch
 - branch equal, branch less than, etc

Which Instructions?

- Arithmetic
 - add, subtract, multiply, divide
- Logical
 - and, or, shift left, shift right
- Data transfer
 - load word, store word
- Conditional branch
 - branch equal, branch less than, etc
- Unconditional branch (jump)

Which Instructions?

- Arithmetic
 - add, subtract, multiply, divide
- Logical
 - and, or, shift left, shift right
- Data transfer
 - load word, store word
- Conditional branch
 - branch equal, branch less than, etc
- Unconditional branch (jump)
 - jump, jump and link, return

Which Instructions?

- Arithmetic
 - add, subtract, multiply, divide
- Logical
 - and, or, shift left, shift right
- Data transfer
 - load word, store word
- Conditional branch
 - branch equal, branch less than, etc
- Unconditional branch (jump) **Control**
 - jump, jump and link, return

Control Flow

Control Flow

- Conditional Branches

Control Flow

- Conditional Branches
 - if-then-else, loops, switches

Control Flow

- Conditional Branches
 - if-then-else, loops, switches
- Unconditional Branches (Jumps)

Control Flow

- Conditional Branches
 - if-then-else, loops, switches
- Unconditional Branches (Jumps)
 - Goto-like Jumps/Branches

Control Flow

- Conditional Branches
 - if-then-else, loops, switches
- Unconditional Branches (Jumps)
 - Goto-like Jumps/Branches
 - Function/Procedure calls

Control Flow

- Conditional Branches
 - if-then-else, loops, switches
- Unconditional Branches (Jumps)
 - Goto-like Jumps/Branches
 - Function/Procedure calls
- A conditional branch must specify two things
 - Location that the branch jumps to if taken (target)
 - Condition under which the branch is taken

Specifying Branch Target

Specifying Branch Target

- Studies show that almost all conditional branches go short distances from the current program counter (loops, if-then-else). **Is this a good thing? Why/Why not?**

Specifying Branch Target

- Studies show that almost all conditional branches go short distances from the current program counter (loops, if-then-else). **Is this a good thing? Why/Why not?**
 - We can specify a relative address in much fewer bits than an absolute address

Specifying Branch Target

- Studies show that almost all conditional branches go short distances from the current program counter (loops, if-then-else). **Is this a good thing? Why/Why not?**
 - We can specify a relative address in much fewer bits than an absolute address
 - e.g., `beq $1, $2, 100` $\Rightarrow$ if ($\$1 == \$2$) $PC = (PC+4) + 100 * 4$

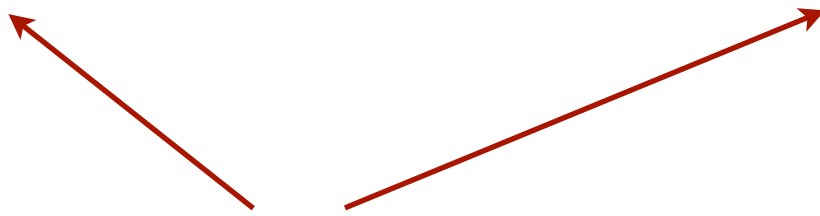
Specifying Branch Target

- Studies show that almost all conditional branches go short distances from the current program counter (loops, if-then-else). **Is this a good thing? Why/Why not?**

- We can specify a relative address in much fewer bits than an absolute address

- e.g., `beq $1, $2, 100` $\Rightarrow$ if ($\$1 == \$2$) $PC = (PC+4) + 100 * 4$

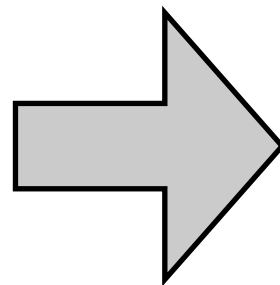
condition



Specifying Branch's Target

- beq, bne beq r1, r2, addr => if (r1 == r2) goto addr
- slt \$1, \$2, \$3 => if (\$2 < \$3) \$1 = 1; else \$1 = 0
- these, combined with \$0, can implement all fundamental branch conditions
- Always, never, !=, ==, >, <=, >=, <, >(unsigned), <= (unsigned), ...

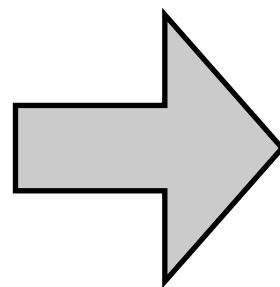
```
if (i<j)
    w = w+1;
else
    w = 5;
```



Specifying Branch's Target

- beq, bne beq r1, r2, addr => if (r1 == r2) goto addr
- slt \$1, \$2, \$3 => if (\$2 < \$3) \$1 = 1; else \$1 = 0
- these, combined with \$0, can implement all fundamental branch conditions
- Always, never, !=, ==, >, <=, >=, <, >(unsigned), <= (unsigned), ...

```
if (i<j)
    w = w+1;
else
    w = 5;
```

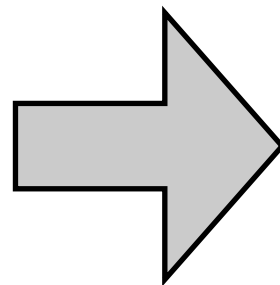


```
slt $temp, $i, $j
beq $temp, $0, L1
add $w, $w, #1
beq $0, $0, L2
L1: add $w, $0, #5
L2:
```

Specifying Branch's Target

- beq, bne beq r1, r2, addr => if (r1 == r2) goto addr
- slt \$1, \$2, \$3 => if (\$2 < \$3) \$1 = 1; else \$1 = 0
- these, combined with \$0, can implement all fundamental branch conditions
- Always, never, !=, ==, >, <=, >=, <, >(unsigned), <= (unsigned), ...

```
if (i<j)
    w = w+1;
else
    w = 5;
```

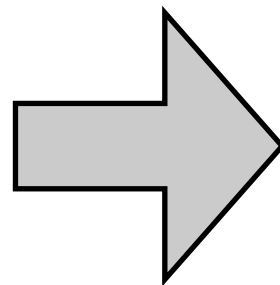


```
slt $temp, $i, $j
beq $temp, $0, L1
add $w, $w, #1
beq $0, $0, L2
L1: add $w, $0, #5
L2:
```

Specifying Branch's Target

- beq, bne beq r1, r2, addr => if (r1 == r2) goto addr
- slt \$1, \$2, \$3 => if (\$2 < \$3) \$1 = 1; else \$1 = 0
- these, combined with \$0, can implement all fundamental branch conditions
- Always, never, !=, ==, >, <=, >=, <, >(unsigned), <= (unsigned), ...

```
if (i<j)
    w = w+1;
else
    w = 5;
```

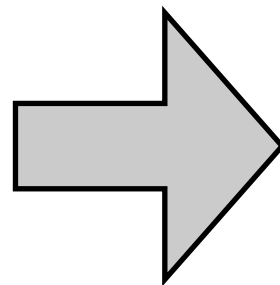


```
slt $temp, $i, $j
beq $temp, $0, L1
add $w, $w, #1
beq $0, $0, L2
L1: add $w, $0, #5
L2:
```

Specifying Branch's Target

- beq, bne beq r1, r2, addr => if (r1 == r2) goto addr
- slt \$1, \$2, \$3 => if (\$2 < \$3) \$1 = 1; else \$1 = 0
- these, combined with \$0, can implement all fundamental branch conditions
- Always, never, !=, ==, >, <=, >=, <, >(unsigned), <= (unsigned), ...

```
if (i<j)
    w = w+1;
else
    w = 5;
```

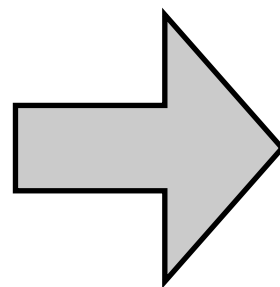


```
slt $temp, $i, $j
beq $temp, $0, L1
add $w, $w, #1
beq $0, $0, L2
L1: add $w, $0, #5
L2:
```

Specifying Branch's Target

- beq, bne beq r1, r2, addr => if (r1 == r2) goto addr
- slt \$1, \$2, \$3 => if (\$2 < \$3) \$1 = 1; else \$1 = 0
- these, combined with \$0, can implement all fundamental branch conditions
- Always, never, !=, ==, >, <=, >=, <, >(unsigned), <= (unsigned), ...

```
if (i<j)
    w = w+1;
else
    w = 5;
```

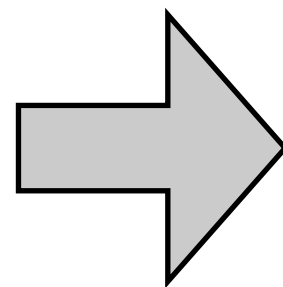


```
slt $temp, $i, $j
beq $temp, $0, L1
add $w, $w, #1
beq $0, $0, L2
L1: add $w, $0, #5
L2:
```

Specifying Branch's Target

- beq, bne beq r1, r2, addr => if (r1 == r2) goto addr
- slt \$1, \$2, \$3 => if (\$2 < \$3) \$1 = 1; else \$1 = 0
- these, combined with \$0, can implement all fundamental branch conditions
- Always, never, !=, ==, >, <=, >=, <, >(unsigned), <= (unsigned), ...

```
if (i<j)
    w = w+1;
else
    w = 5;
```

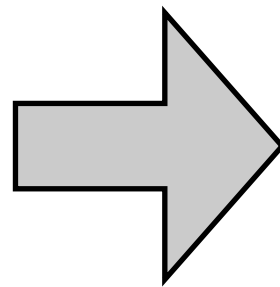


```
slt $temp, $i, $j
beq $temp, $0, L1
add $w, $w, #1
beq $0, $0, L2
L1: add $w, $0, #5
L2:
```

Specifying Branch's Target

- beq, bne beq r1, r2, addr => if (r1 == r2) goto addr
- slt \$1, \$2, \$3 => if (\$2 < \$3) \$1 = 1; else \$1 = 0
- these, combined with \$0, can implement all fundamental branch conditions
- Always, never, !=, ==, >, <=, >=, <, >(unsigned), <= (unsigned), ...

```
if (i<j)
    w = w+1;
else
    w = 5;
```



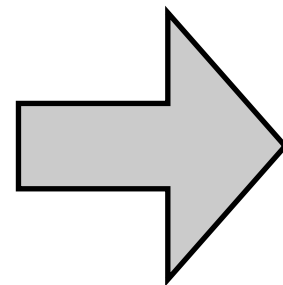
```
slt $temp, $i, $j
beq $temp, $0, L1
add $w, $w, #1
beq $0, $0, L2
L1: add $w, $0, #5
L2:
```

Specifying Branch's Target

- beq, bne beq r1, r2, addr => if (r1 == r2) goto addr
- slt \$1, \$2, \$3 => if (\$2 < \$3) \$1 = 1; else \$1 = 0
- these, combined with \$0, can implement all fundamental branch conditions
- Always, never, !=, ==, >, <=, >=, <, >(unsigned), <= (unsigned), ...

i=5, j=10

```
if (i<j)
    w = w+1;
else
    w = 5;
```



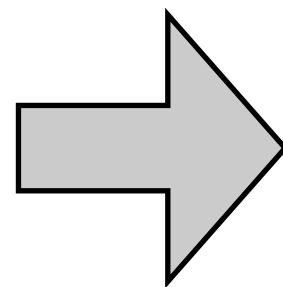
```
slt $temp, $i, $j
beq $temp, $0, L1
add $w, $w, #1
beq $0, $0, L2
L1: add $w, $0, #5
L2:
```

Specifying Branch's Target

- beq, bne beq r1, r2, addr => if (r1 == r2) goto addr
- slt \$1, \$2, \$3 => if (\$2 < \$3) \$1 = 1; else \$1 = 0
- these, combined with \$0, can implement all fundamental branch conditions
- Always, never, !=, ==, >, <=, >=, <, >(unsigned), <= (unsigned), ...

i=5, j=10

```
if (i<j)
    w = w+1;
else
    w = 5;
```



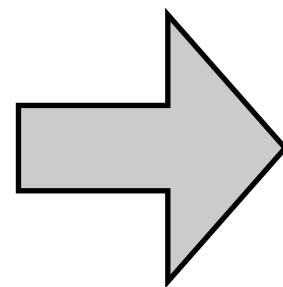
```
slt $temp, $i, $j
beq $temp, $0, L1
add $w, $w, #1
beq $0, $0, L2
L1: add $w, $0, #5
L2:
```

Specifying Branch's Target

- beq, bne beq r1, r2, addr => if (r1 == r2) goto addr
- slt \$1, \$2, \$3 => if (\$2 < \$3) \$1 = 1; else \$1 = 0
- these, combined with \$0, can implement all fundamental branch conditions
- Always, never, !=, ==, >, <=, >=, <, >(unsigned), <= (unsigned), ...

i=5, j=10

```
if (i<j)
    w = w+1;
else
    w = 5;
```



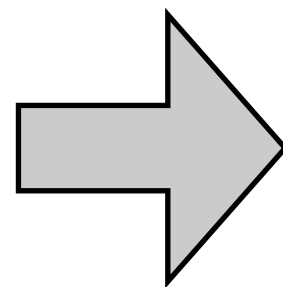
```
slt $temp, $i, $j
beq $temp, $0, L1
add $w, $w, #1
beq $0, $0, L2
L1: add $w, $0, #5
L2:
```

Specifying Branch's Target

- beq, bne beq r1, r2, addr => if (r1 == r2) goto addr
- slt \$1, \$2, \$3 => if (\$2 < \$3) \$1 = 1; else \$1 = 0
- these, combined with \$0, can implement all fundamental branch conditions
- Always, never, !=, ==, >, <=, >=, <, >(unsigned), <= (unsigned), ...

i=5, j=10

```
if (i<j)
    w = w+1;
else
    w = 5;
```



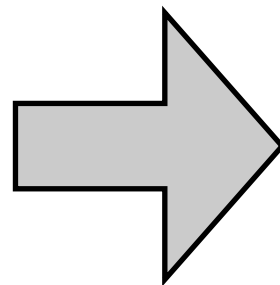
```
slt $temp, $i, $j
beq $temp, $0, L1
add $w, $w, #1
beq $0, $0, L2
L1: add $w, $0, #5
L2:
```

Specifying Branch's Target

- beq, bne beq r1, r2, addr => if (r1 == r2) goto addr
- slt \$1, \$2, \$3 => if (\$2 < \$3) \$1 = 1; else \$1 = 0
- these, combined with \$0, can implement all fundamental branch conditions
- Always, never, !=, ==, >, <=, >=, <, >(unsigned), <= (unsigned), ...

i=5, j=10

```
if (i<j)
    w = w+1;
else
    w = 5;
```



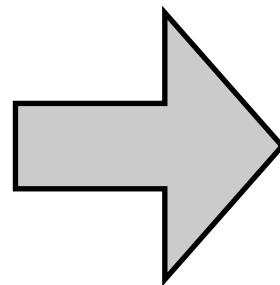
```
slt $temp, $i, $j
beq $temp, $0, L1
add $w, $w, #1
beq $0, $0, L2
L1: add $w, $0, #5
L2:
```

Specifying Branch's Target

- beq, bne beq r1, r2, addr => if (r1 == r2) goto addr
- slt \$1, \$2, \$3 => if (\$2 < \$3) \$1 = 1; else \$1 = 0
- these, combined with \$0, can implement all fundamental branch conditions
- Always, never, !=, ==, >, <=, >=, <, >(unsigned), <= (unsigned), ...

i=5, j=10

```
if (i<j)
    w = w+1;
else
    w = 5;
```



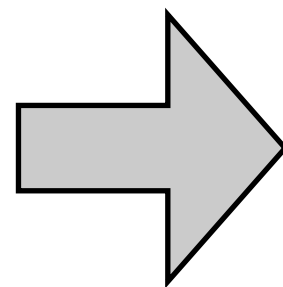
```
slt $temp, $i, $j
beq $temp, $0, L1
add $w, $w, #1
beq $0, $0, L2
L1: add $w, $0, #5
L2:
```

Specifying Branch's Target

- beq, bne beq r1, r2, addr => if (r1 == r2) goto addr
- slt \$1, \$2, \$3 => if (\$2 < \$3) \$1 = 1; else \$1 = 0
- these, combined with \$0, can implement all fundamental branch conditions
- Always, never, !=, ==, >, <=, >=, <, >(unsigned), <= (unsigned), ...

i=6, j=4

```
if (i<j)
    w = w+1;
else
    w = 5;
```



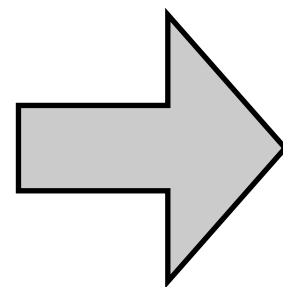
```
slt $temp, $i, $j
beq $temp, $0, L1
add $w, $w, #1
beq $0, $0, L2
L1: add $w, $0, #5
L2:
```

Specifying Branch's Target

- beq, bne beq r1, r2, addr => if (r1 == r2) goto addr
- slt \$1, \$2, \$3 => if (\$2 < \$3) \$1 = 1; else \$1 = 0
- these, combined with \$0, can implement all fundamental branch conditions
- Always, never, !=, ==, >, <=, >=, <, >(unsigned), <= (unsigned), ...

i=6, j=4

```
if (i<j)
    w = w+1;
else
    w = 5;
```



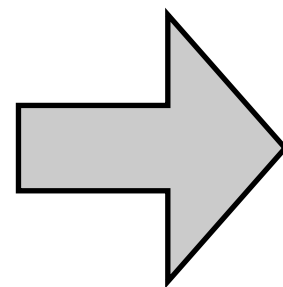
```
slt $temp, $i, $j
beq $temp, $0, L1
add $w, $w, #1
beq $0, $0, L2
L1: add $w, $0, #5
L2:
```

Specifying Branch's Target

- beq, bne beq r1, r2, addr => if (r1 == r2) goto addr
- slt \$1, \$2, \$3 => if (\$2 < \$3) \$1 = 1; else \$1 = 0
- these, combined with \$0, can implement all fundamental branch conditions
- Always, never, !=, ==, >, <=, >=, <, >(unsigned), <= (unsigned), ...

i=6, j=4

```
if (i<j)
    w = w+1;
else
    w = 5;
```



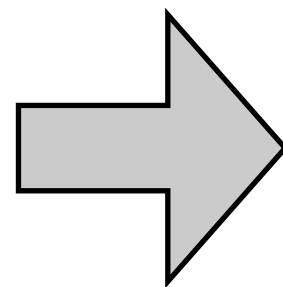
```
slt $temp, $i, $j
beq $temp, $0, L1
add $w, $w, #1
beq $0, $0, L2
L1: add $w, $0, #5
L2:
```

Specifying Branch's Target

- beq, bne beq r1, r2, addr => if (r1 == r2) goto addr
- slt \$1, \$2, \$3 => if (\$2 < \$3) \$1 = 1; else \$1 = 0
- these, combined with \$0, can implement all fundamental branch conditions
- Always, never, !=, ==, >, <=, >=, <, >(unsigned), <= (unsigned), ...

i=6, j=4

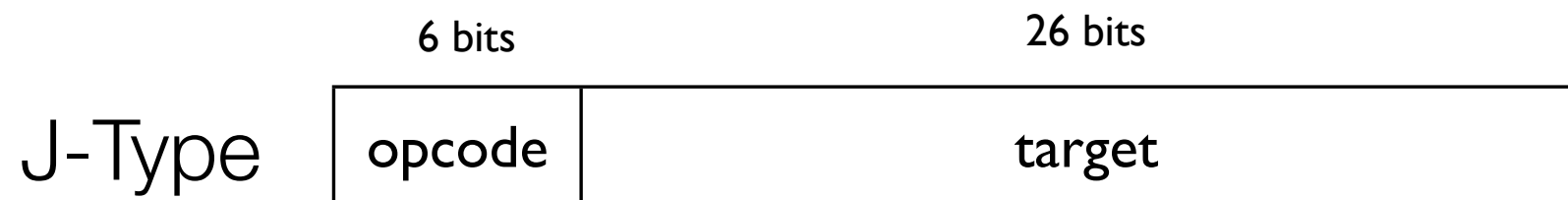
```
if (i<j)
    w = w+1;
else
    w = 5;
```



```
slt $temp, $i, $j
beq $temp, $0, L1
add $w, $w, #1
beq $0, $0, L2
L1: add $w, $0, #5
L2:
```

Jumps (Unconditional Branches)

- need to be able to jump to an absolute address sometime
- need to be able to do procedure calls and return
- jump -- j 10000 => PC = 10000
- jump and link -- jal 100000 => \$31 = PC + 4; PC = 10000
 - used for procedure calls (\$31=\$ra)



- jump register -- jr \$31 => PC = \$31
- used for returns, but can be useful for lots of other things.

Branch and Jump Addressing Mode

Branch and Jump Addressing Mode

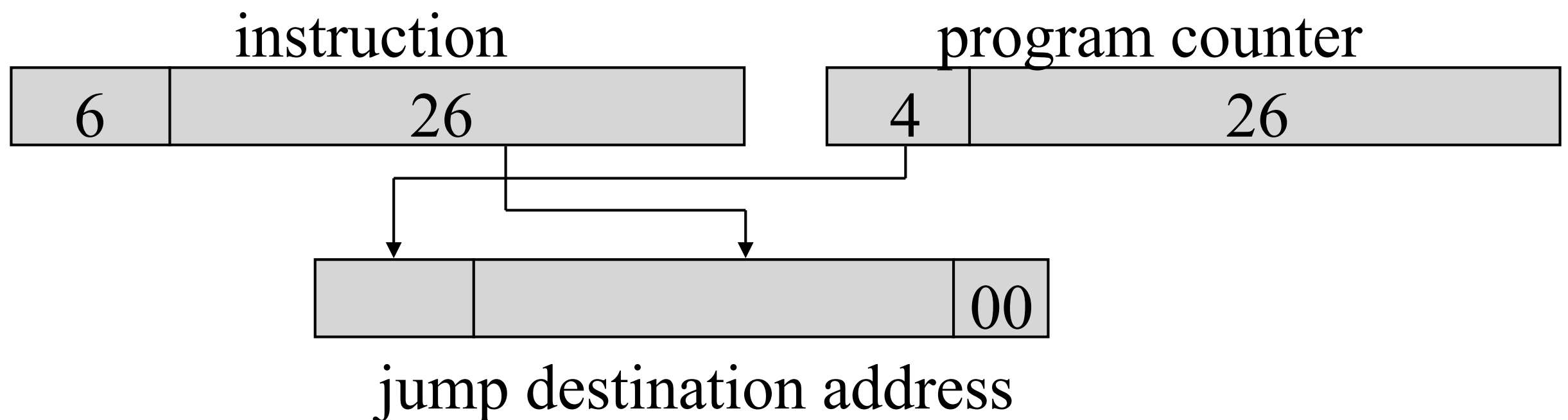
- Branch (e.g., beq) uses PC-relative addressing mode (uses few bits if address typically close). That is, it uses base+displacement mode, with the PC being the base. If opcode is 6 bits, how many bits are available for displacement? Determines how far can you jump.

Branch and Jump Addressing Mode

- Branch (e.g., beq) uses PC-relative addressing mode (uses few bits if address typically close). That is, it uses base+displacement mode, with the PC being the base. If opcode is 6 bits, how many bits are available for displacement? Determines how far can you jump.
- Jump uses pseudo-direct addressing mode. 26 bits of the address is in the instruction, the rest is taken from the PC.

Branch and Jump Addressing Mode

- Branch (e.g., beq) uses PC-relative addressing mode (uses few bits if address typically close). That is, it uses base+displacement mode, with the PC being the base. If opcode is 6 bits, how many bits are available for displacement? Determines how far can you jump.
- Jump uses pseudo-direct addressing mode. 26 bits of the address is in the instruction, the rest is taken from the PC.



Summary of MIPS

MIPS operands

Name	Example	Comments
32 registers	<code>\$s0-\$s7, \$t0-\$t9, \$zero, \$a0-\$a3, \$v0-\$v1, \$gp, \$fp, \$sp, \$ra</code>	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register <code>\$zero</code> always equals 0. <code>\$gp</code> (28) is the global pointer, <code>\$sp</code> (29) is the stack pointer, <code>\$fp</code> (30) is the frame pointer, and <code>\$ra</code> (31) is the return address.
2^{30} memory words	<code>Memory[0], Memory[4], . . . , Memory[4294967292]</code>	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	<code>add \$s1,\$s2,\$s3</code>	<code>\$s1 = \$s2 + \$s3</code>	three register operands
	subtract	<code>sub \$s1,\$s2,\$s3</code>	<code>\$s1 = \$s2 - \$s3</code>	three register operands
Data transfer	load word	<code>lw \$s1,100(\$s2)</code>	<code>\$s1 = Memory[\$s2 + 100]</code>	Data from memory to register
	store word	<code>sw \$s1,100(\$s2)</code>	<code>Memory[\$s2 + 100] = \$s1</code>	Data from register to memory
Logical	and	<code>and \$s1,\$s2,\$s3</code>	<code>\$s1 = \$s2 & \$s3</code>	three reg. operands; bit-by-bit AND
	or	<code>or \$s1,\$s2,\$s3</code>	<code>\$s1 = \$s2 \$s3</code>	three reg. operands; bit-by-bit OR
	nor	<code>nor \$s1,\$s2,\$s3</code>	<code>\$s1 = ~ (\$s2 \$s3)</code>	three reg. operands; bit-by-bit NOR
	and immediate	<code>andi \$s1,\$s2,100</code>	<code>\$s1 = \$s2 & 100</code>	Bit-by-bit AND reg with constant
	or immediate	<code>ori \$s1,\$s2,100</code>	<code>\$s1 = \$s2 100</code>	Bit-by-bit OR reg with constant
	shift left logical	<code>sll \$s1,\$s2,10</code>	<code>\$s1 = \$s2 << 10</code>	Shift left by constant
	shift right logical	<code>srl \$s1,\$s2,10</code>	<code>\$s1 = \$s2 >> 10</code>	Shift right by constant
Conditional branch	branch on equal	<code>beq \$s1,\$s2,L</code>	if (<code>\$s1 == \$s2</code>) go to L	Equal test and branch
	branch on not equal	<code>bne \$s1,\$s2,L</code>	if (<code>\$s1 != \$s2</code>) go to L	Not equal test and branch
	set on less than	<code>slt \$s1,\$s2,\$s3</code>	if (<code>\$s2 < \$s3</code>) <code>\$s1 = 1</code> ; else <code>\$s1 = 0</code>	Compare less than; used with <code>beq</code> , <code>bne</code>
	set on less than immediate	<code>slt \$s1,\$s2,100</code>	if (<code>\$s2 < 100</code>) <code>\$s1 = 1</code> ; else <code>\$s1 = 0</code>	Compare less than immediate; used with <code>beq</code> , <code>bne</code>
Unconditional jump	jump	<code>j L</code>	go to L	Jump to target address
	jump register	<code>jr \$ra</code>	go to <code>\$ra</code>	For procedure return
	jump and link	<code>jal L</code>	<code>\$ra = PC + 4</code> ; go to L	For procedure call

Practice

- Translate the C code into assembly:

```
for(i = 0; i < 100; i++)  
{  
    sum+=A[i];  
}
```

Practice

- Translate the C code into assembly:

```
for(i = 0; i < 100; i++)  
{  
    sum+=A[i];  
}
```

1. initialization
2. load A[i] from memory to register
3. add the value of A[i] to sum
4. increase by 1
5. check if i still < 100

Practice

- Translate the C code into assembly:

```
for(i = 0; i < 100; i++)  
{  
    sum+=A[i];  
}
```

1. initialization
2. load A[i] from memory to register
3. add the value of A[i] to sum
4. increase by 1
5. check if i still < 100

Assume
int is 32 bits
\$s0 = &A[0]
\$v0 = sum;
\$t0 = i;

Practice

- Translate the C code into assembly:

```
for(i = 0; i < 100; i++)  
{  
    sum+=A[i];  
}
```



1. initialization
2. load A[i] from memory to register
3. add the value of A[i] to sum
4. increase by 1
5. check if i still < 100

Assume
int is 32 bits
\$s0 = &A[0]
\$v0 = sum;
\$t0 = i;

Practice

- Translate the C code into assembly:

```
for(i = 0; i < 100; i++)          and $t0, $t0, $zero
{
    sum+=A[i];
}
```



1. initialization
2. load A[i] from memory to register
3. add the value of A[i] to sum
4. increase by 1
5. check if i still < 100

Assume
int is 32 bits
\$s0 = &A[0]
\$v0 = sum;
\$t0 = i;

Practice

- Translate the C code into assembly:

```
for(i = 0; i < 100; i++)  
{  
    sum+=A[i];  
}
```



```
and    $t0, $t0, $zero  
addi   $t1, $zero, 100
```

1. initialization
2. load A[i] from memory to register
3. add the value of A[i] to sum
4. increase by 1
5. check if i still < 100

Assume
int is 32 bits
\$s0 = &A[0]
\$v0 = sum;
\$t0 = i;

Practice

- Translate the C code into assembly:

```
for(i = 0; i < 100; i++)  
{  
    sum+=A[i];  
}
```



```
and    $t0, $t0, $zero  
addi   $t1, $zero, 100  
lw     $t3, 0($s0)
```

1. initialization
2. load A[i] from memory to register
3. add the value of A[i] to sum
4. increase by 1
5. check if i still < 100

Assume
int is 32 bits
\$s0 = &A[0]
\$v0 = sum;
\$t0 = i;

Practice

- Translate the C code into assembly:

```
for(i = 0; i < 100; i++)  
{  
    sum+=A[i];  
}
```



```
and    $t0, $t0, $zero  
addi   $t1, $zero, 100  
lw     $t3, 0($s0)  
add    $v0, $v0, $t3
```

1. initialization
2. load A[i] from memory to register
3. add the value of A[i] to sum
4. increase by 1
5. check if i still < 100

Assume
int is 32 bits
\$s0 = &A[0]
\$v0 = sum;
\$t0 = i;

Practice

- Translate the C code into assembly:

```
for(i = 0; i < 100; i++)  
{  
    sum+=A[i];  
}
```



```
and    $t0, $t0, $zero  
addi   $t1, $zero, 100  
lw     $t3, 0($s0)  
add    $v0, $v0, $t3  
addi   $s0, $s0, 4
```

1. initialization
2. load A[i] from memory to register
3. add the value of A[i] to sum
4. increase by 1
5. check if i still < 100

Assume
int is 32 bits
\$s0 = &A[0]
\$v0 = sum;
\$t0 = i;

Practice

- Translate the C code into assembly:

```
for(i = 0; i < 100; i++)  
{  
    sum+=A[i];  
}
```



```
and    $t0, $t0, $zero  
addi   $t1, $zero, 100  
lw     $t3, 0($s0)  
add    $v0, $v0, $t3  
addi   $s0, $s0, 4  
addi   $t0, $t0, 1
```

1. initialization
2. load A[i] from memory to register
3. add the value of A[i] to sum
4. increase by 1
5. check if i still < 100

Assume
int is 32 bits
\$s0 = &A[0]
\$v0 = sum;
\$t0 = i;

Practice

- Translate the C code into assembly:

```
for(i = 0; i < 100; i++)  
{  
    sum+=A[i];  
}
```



```
and    $t0, $t0, $zero  
addi   $t1, $zero, 100  
lw     $t3, 0($s0)  
add    $v0, $v0, $t3  
addi   $s0, $s0, 4  
addi   $t0, $t0, 1  
bne    $t1, $t0, LOOP
```

1. initialization
2. load A[i] from memory to register
3. add the value of A[i] to sum
4. increase by 1
5. check if i still < 100

Assume
int is 32 bits
\$s0 = &A[0]
\$v0 = sum;
\$t0 = i;

Practice

- Translate the C code into assembly:

```
for(i = 0; i < 100; i++)  
{  
    sum+=A[i];  
}
```



```
and    $t0, $t0, $zero  
addi   $t1, $zero, 100  
lw     $t3, 0($s0)  
add    $v0, $v0, $t3  
addi   $s0, $s0, 4  
addi   $t0, $t0, 1  
bne    $t1, $t0, LOOP
```

1. initialization
2. load A[i] from memory to register
3. add the value of A[i] to sum
4. increase by 1
5. check if i still < 100

Assume
int is 32 bits
\$s0 = &A[0]
\$v0 = sum;
\$t0 = i;

Practice

- Translate the C code into assembly:

<pre>for(i = 0; i < 100; i++) { sum+=A[i]; }</pre>		<pre>LOOP: and \$t0, \$t0, \$zero addi \$t1, \$zero, 100 lw \$t3, 0(\$s0) add \$v0, \$v0, \$t3 addi \$s0, \$s0, 4 addi \$t0, \$t0, 1 bne \$t1, \$t0, LOOP</pre>
---	---	--

1. initialization
2. load A[i] from memory to register
3. add the value of A[i] to sum
4. increase by 1
5. check if i still < 100

Assume
int is 32 bits
\$s0 = &A[0]
\$v0 = sum;
\$t0 = i;

Practice

- Translate the C code into assembly:

```
for(i = 0; i < 100; i++)  
{  
    sum+=A[i];  
}
```

label

→

LOOP:

```
and    $t0, $t0, $zero  
addi   $t1, $zero, 100  
lw     $t3, 0($s0)  
add    $v0, $v0, $t3  
addi   $s0, $s0, 4  
addi   $t0, $t0, 1  
bne    $t1, $t0, LOOP
```

1. initialization
2. load A[i] from memory to register
3. add the value of A[i] to sum
4. increase by 1
5. check if i still < 100

Assume
int is 32 bits
\$s0 = &A[0]
\$v0 = sum;
\$t0 = i;

Practice

- Translate the C code into assembly:

```
for(i = 0; i < 100; i++)  
{  
    sum+=A[i];  
}
```

label

LOOP:

```
and    $t0, $t0, $zero    #let i = 0  
addi   $t1, $zero, 100    #temp = 100  
lw     $t3, 0($s0)        #temp1 = A[i]  
add    $v0, $v0, $t3      #sum += temp1  
addi   $s0, $s0, 4        #addr of A[i+1]  
addi   $t0, $t0, 1        #i = i+1  
bne    $t1, $t0, LOOP     #if i < 100
```

1. initialization
2. load A[i] from memory to register
3. add the value of A[i] to sum
4. increase by 1
5. check if i still < 100

Assume
int is 32 bits
\$s0 = &A[0]
\$v0 = sum;
\$t0 = i;