Math 230
Assembly Programming
(*AKA Computer Organization*)
Spring 2008

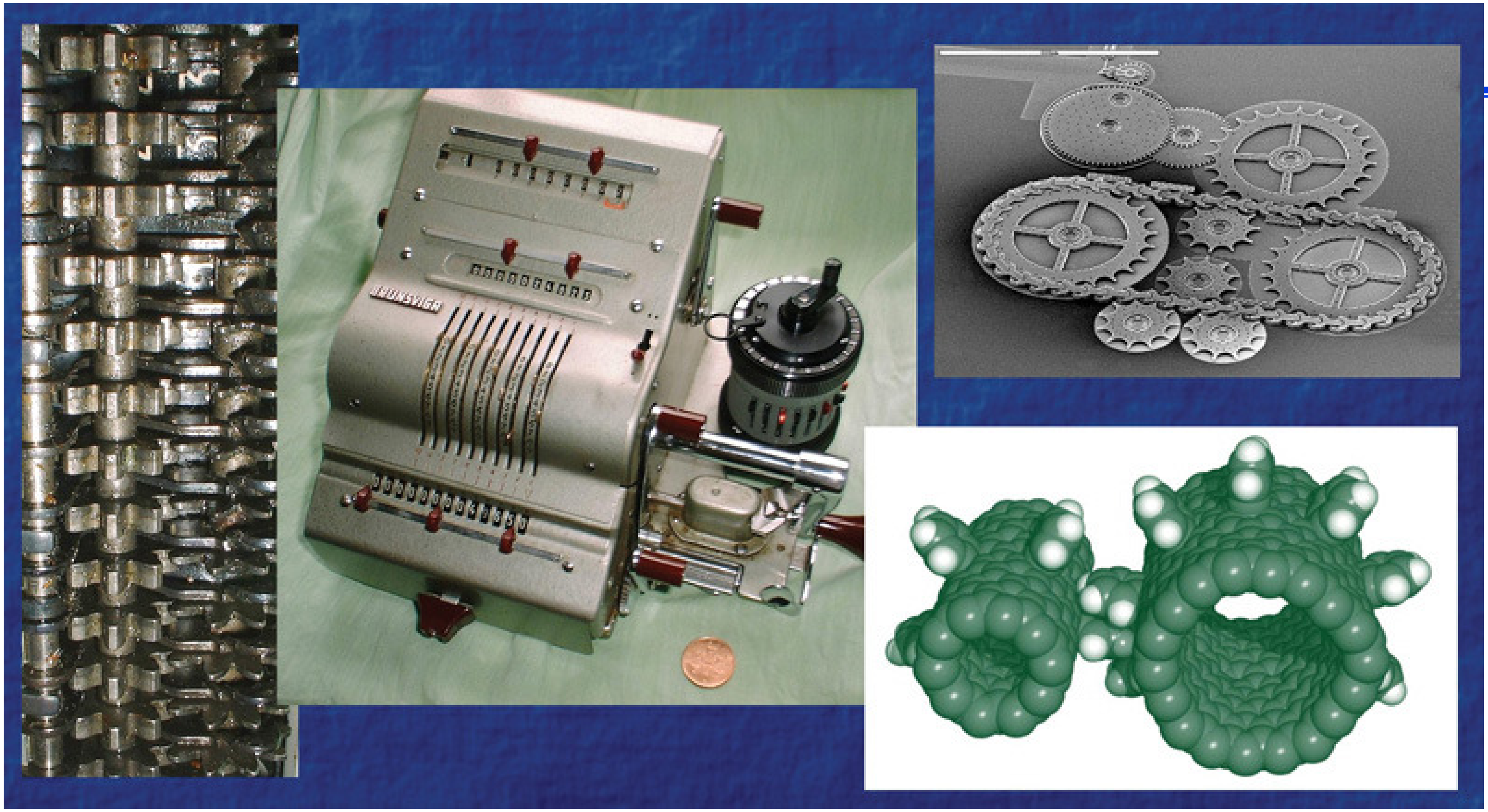
MIPS Intro

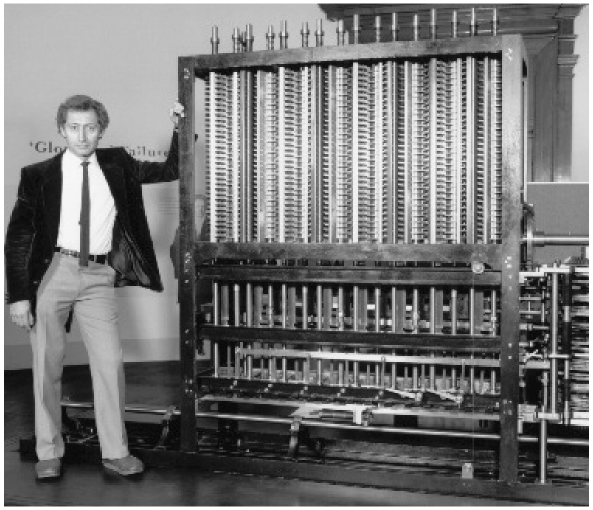
Adapted from slides developed for:

Mary J. Irwin PSU CSE331

Dave Patterson's UCB CS152

MIPS - originally an acronym for **Microprocessor without Interlocked Pipeline Stages**)


BRUNSVIGA



Below the Program

Below the Program

- Machine (object) code (for MIPS)

```
000000 00000 00101 0001000010000000
000000 00100 00010 0001000000100000
 . . .
```

Below the Program

- Assembly language program (for MIPS)

```
swap:  sll    $2, $5, 2
       add    $2, $4,$2
       lw     $15, 0($2)
       lw     $16, 4($2)
       sw     $16, 0($2)
       sw     $15, 4($2)
       jr     $31
```

- Machine (object) code (for MIPS)

```
000000 00000 00101 0001000010000000
000000 00100 00010 0001000000100000
 .  .  .
```

Below the Program

- High-level language program (in C)

```
swap (int v[], int k)
{int temp;
     temp = v[k];
     v[k] = v[k+1];
     v[k+1] = temp;
}
```

- Assembly language program (for MIPS)

```
swap:  sll    $2, $5, 2
       add    $2, $4,$2
       lw     $15, 0($2)
       lw     $16, 4($2)
       sw     $16, 0($2)
       sw     $15, 4($2)
       jr     $31
```

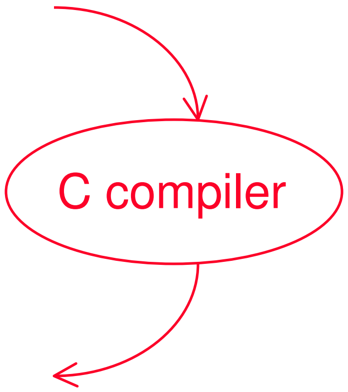
- Machine (object) code (for MIPS)

```
000000 00000 00101 0001000010000000
000000 00100 00010 0001000000100000
 . . .
```

Below the Program

- High-level language program (in C)

```
swap (int v[], int k)
{int temp;
     temp = v[k];
     v[k] = v[k+1];
     v[k+1] = temp;
}
```



C compiler

- Assembly language program (for MIPS)

```
swap: sll    $2, $5, 2
      add    $2, $4,$2
      lw     $15, 0($2)
      lw     $16, 4($2)
      sw     $16, 0($2)
      sw     $15, 4($2)
      jr     $31
```

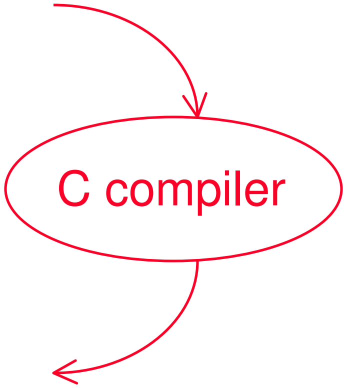
- Machine (object) code (for MIPS)

```
000000 00000 00101 0001000010000000
000000 00100 00010 0001000000100000
 . . .
```

Below the Program

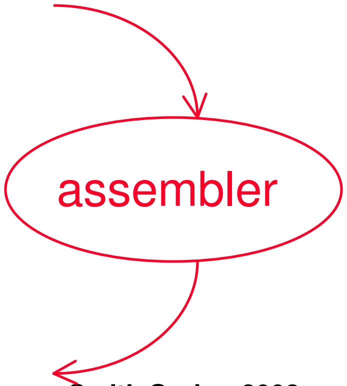
- High-level language program (in C)

```
swap (int v[], int k)
{int temp;
       temp = v[k];
       v[k] = v[k+1];
       v[k+1] = temp;
}
```


C compiler

- Assembly language program (for MIPS)

```
swap:  sll     $2, $5, 2
       add     $2, $4,$2
       lw      $15, 0($2)
       lw      $16, 4($2)
       sw      $16, 0($2)
       sw      $15, 4($2)
       jr      $31
```


assembler

- Machine (object) code (for MIPS)

```
000000 00000 00101 0001000010000000
000000 00100 00010 0001000000100000
 . . .
```

Advantages of Higher-Level Languages

- Higher-level languages

- As a result, very little programming is done today at the assembler level

Advantages of Higher-Level Languages

- Higher-level languages
  - Allow the programmer to think in a more natural language and for their intended use (Fortran for scientific computation, Cobol for business programming, Lisp for symbol manipulation, …)

- As a result, very little programming is done today at the assembler level

Advantages of Higher-Level Languages

- Higher-level languages
  - Allow the programmer to think in a more natural language and for their intended use (Fortran for scientific computation, Cobol for business programming, Lisp for symbol manipulation, …)
  - Improve programmer productivity – more understandable code that is easier to debug and validate
- As a result, very little programming is done today at the assembler level

Advantages of Higher-Level Languages

- Higher-level languages
  - Allow the programmer to think in a more natural language and for their intended use (Fortran for scientific computation, Cobol for business programming, Lisp for symbol manipulation, …)
  - Improve programmer productivity – more understandable code that is easier to debug and validate
  - Improve program maintainability

- As a result, very little programming is done today at the assembler level

Advantages of Higher-Level Languages

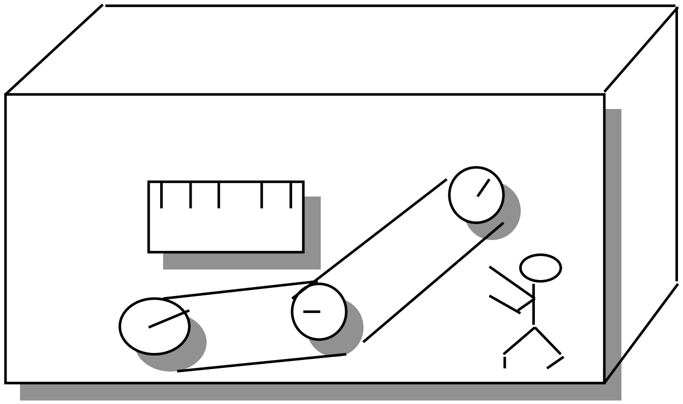
- Higher-level languages
  - Allow the programmer to think in a more natural language and for their intended use (Fortran for scientific computation, Cobol for business programming, Lisp for symbol manipulation, …)
  - Improve programmer productivity – more understandable code that is easier to debug and validate
  - Improve program maintainability
  - Allow programmers to be independent of the computer on which they are developed (compilers and assemblers can translate high-level language programs to the binary instructions of any machine)

- As a result, very little programming is done today at the assembler level

Advantages of Higher-Level Languages

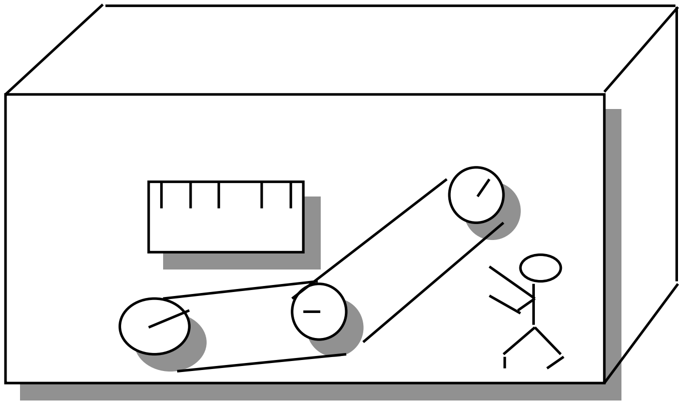
- Higher-level languages
  - Allow the programmer to think in a more natural language and for their intended use (Fortran for scientific computation, Cobol for business programming, Lisp for symbol manipulation, …)
  - Improve programmer productivity – more understandable code that is easier to debug and validate
  - Improve program maintainability
  - Allow programmers to be independent of the computer on which they are developed (compilers and assemblers can translate high-level language programs to the binary instructions of any machine)
  - Emergence of optimizing compilers that produce very efficient assembly code optimized for the target machine
- As a result, very little programming is done today at the assembler level

Machine Organization



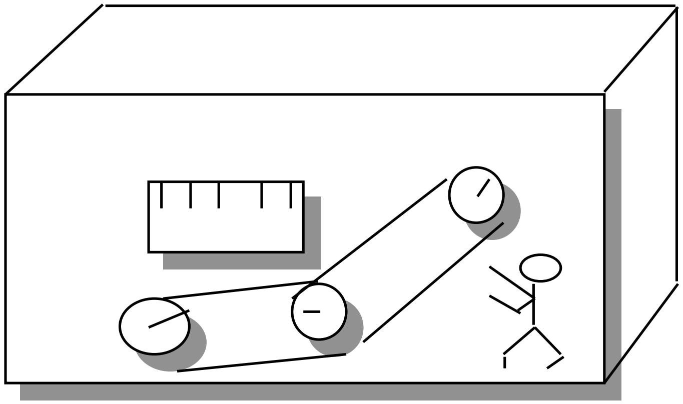
Machine Organization

- Capabilities and performance characteristics of the principal Functional Units (FUs)
  - e.g., register file, ALU, multiplexors, memories, ...



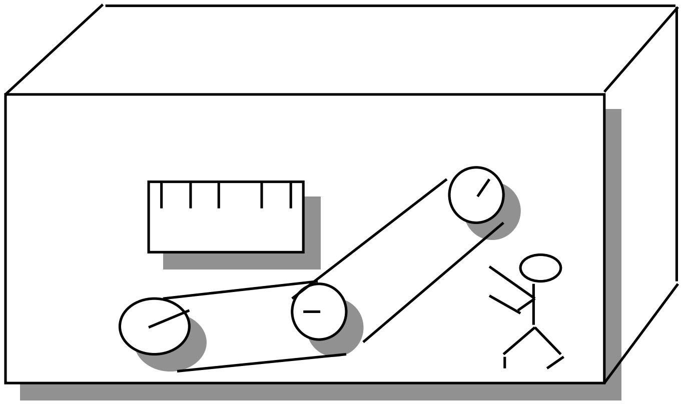
Machine Organization

- Capabilities and performance characteristics of the principal Functional Units (FUs)
  - e.g., register file, ALU, multiplexors, memories, ...
- The ways those FUs are interconnected
  - e.g., buses



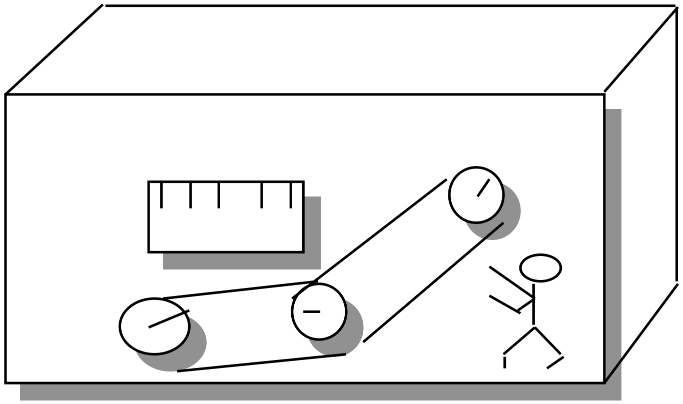
Machine Organization

- Capabilities and performance characteristics of the principal Functional Units (FUs)
  - e.g., register file, ALU, multiplexors, memories, ...
- The ways those FUs are interconnected
  - e.g., buses
- Logic and means by which information flow between FUs is controlled



Machine Organization

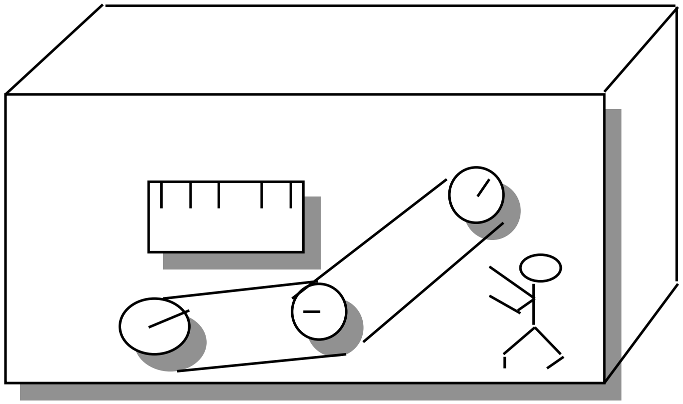
- ❑ Capabilities and performance characteristics of the principal Functional Units (FUs)
  - e.g., register file, ALU, multiplexors, memories, ...
- ❑ The ways those FUs are interconnected
  - e.g., buses
- ❑ Logic and means by which information flow between FUs is controlled



- ❑ The machine’s Instruction Set Architecture (ISA)

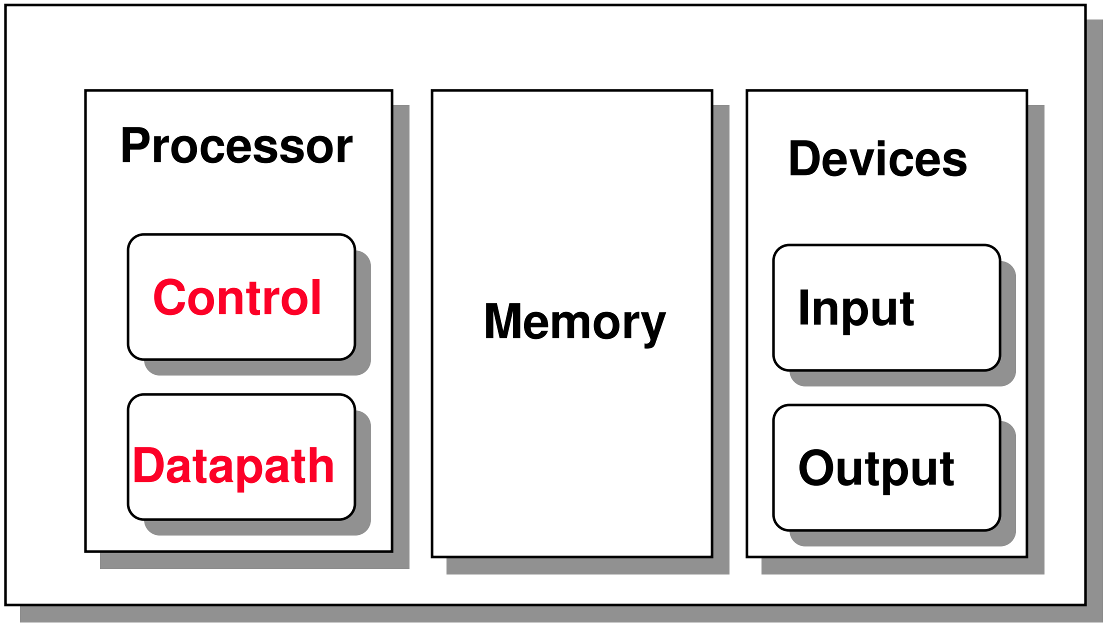
Machine Organization

- Capabilities and performance characteristics of the principal Functional Units (FUs)
  - e.g., register file, ALU, multiplexors, memories, ...
- The ways those FUs are interconnected
  - e.g., buses
- Logic and means by which information flow between FUs is controlled



- The machine's Instruction Set Architecture (ISA)
- Register Transfer Level (RTL) machine description

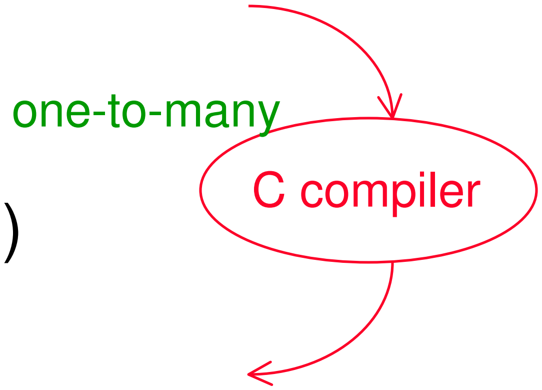
Major Components of a Computer



Below the Program

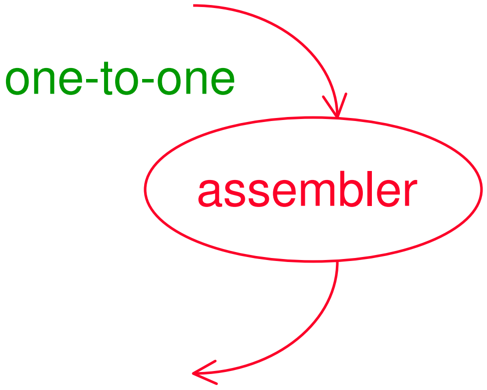
- High-level language program (in C)

```
swap (int v[], int k)
. . .
```



- Assembly language program (for MIPS)

```
swap:   sll     $2, $5, 2
        add     $2, $4, $2
        lw      $15, 0($2)
        lw      $16, 4($2)
        sw      $16, 0($2)
        sw      $15, 4($2)
        jr      $31
```



- Machine (object) code (for MIPS)

```
000000 00000 00101 0001000010000000
000000 00100 00010 0001000000100000
100011 00010 01111 0000000000000000
100011 00010 10000 0000000000000100
101011 00010 10000 0000000000000000
101011 00010 01111 0000000000000100
000000 11111 00000 0000000000001000
```

Below the Program

- High-level language program (in C)

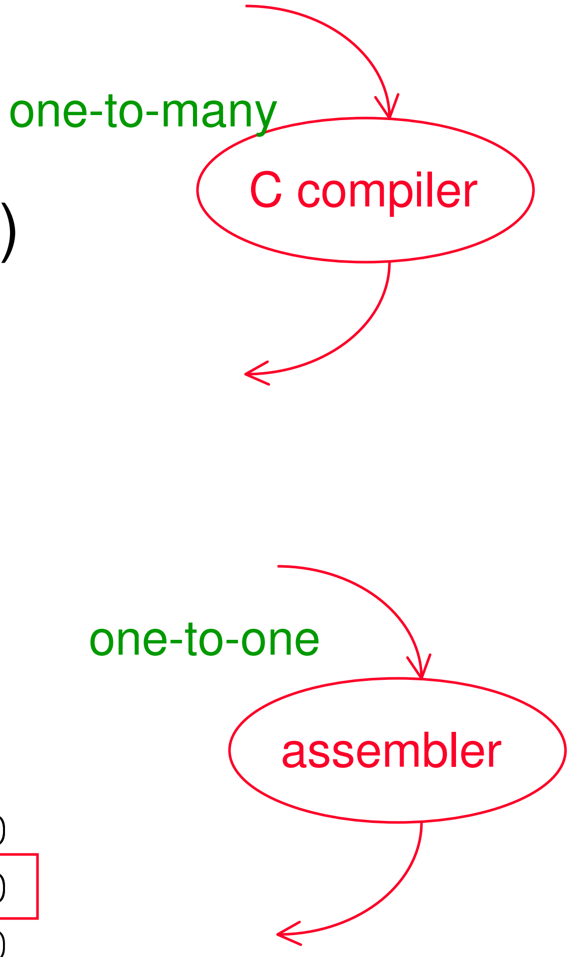
```
swap (int v[], int k)
. . .
```

- Assembly language program (for MIPS)

```
swap:   sll     $2, $5, 2
        add     $2, $4, $2
        lw      $15, 0($2)
        lw      $16, 4($2)
        sw      $16, 0($2)
        sw      $15, 4($2)
        jr      $31
```

- Machine (object) code (for MIPS)

```
000000 00000 00101 0001000010000000
000000 00100 00010 0001000000100000
100011 00010 01111 0000000000000000
100011 00010 10000 0000000000000100
101011 00010 10000 0000000000000000
101011 00010 01111 0000000000000100
000000 11111 00000 0000000000001000
```



one-to-many

C compiler

one-to-one

assembler

Below the Program

- High-level language program (in C)

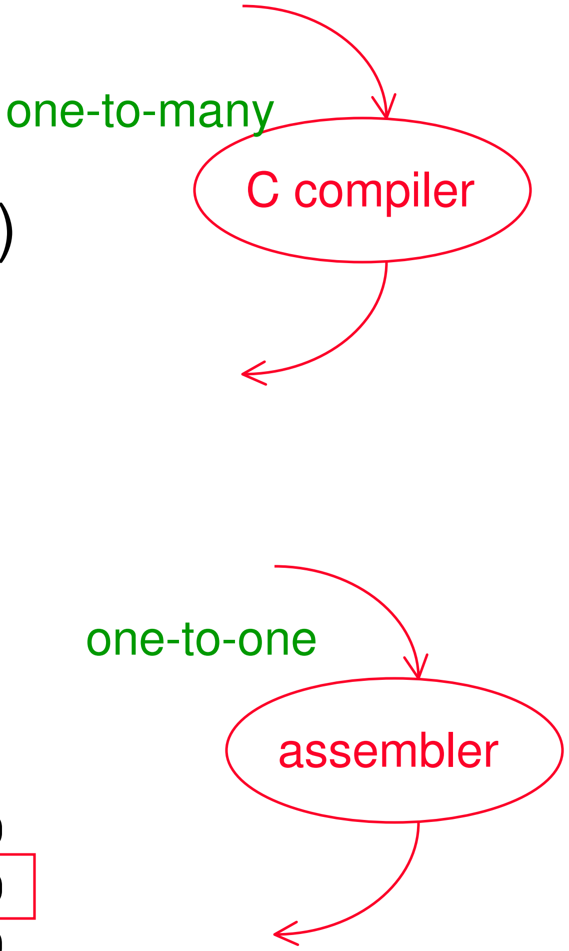
```
swap (int v[], int k)
. . .
```

- Assembly language program (for MIPS)

```
swap:   sll     $2, $5, 2
        add     $2, $4, $2
        lw      $15, 0($2)
        lw      $16, 4($2)
        sw      $16, 0($2)
        sw      $15, 4($2)
        jr      $31
```

- Machine (object) code (for MIPS)

```
000000 00000 00101 0001000010000000
000000 00100 00010 0001000000100000
100011 00010 01111 0000000000000000
100011 00010 10000 0000000000000100
101011 00010 10000 0000000000000000
101011 00010 01111 0000000000000100
000000 11111 00000 0000000000001000
```



Below the Program

- High-level language program (in C)

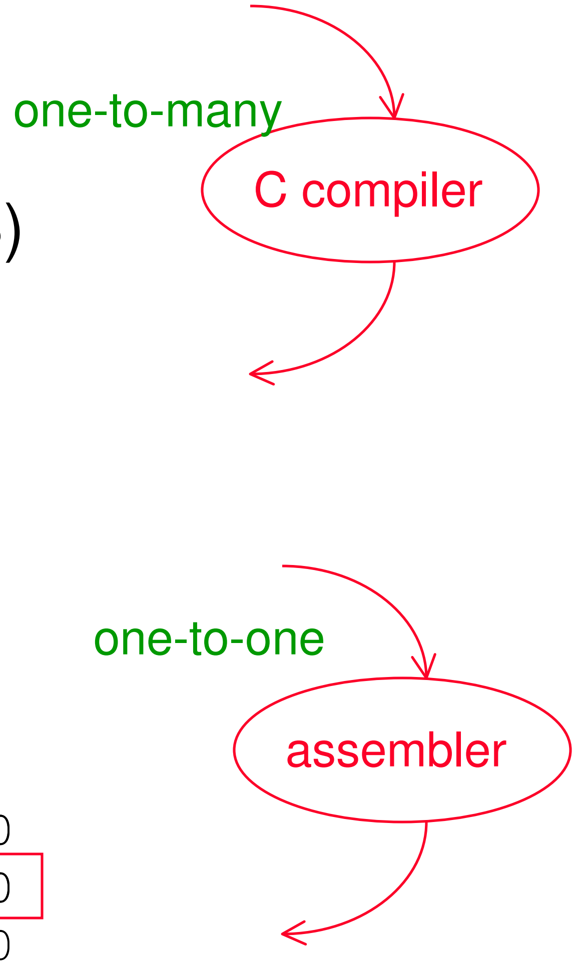
```
swap (int v[], int k)
. . .
```

- Assembly language program (for MIPS)

```
swap:   sll     $2, $5, 2
        add     $2, $4, $2
        lw      $15, 0($2)
        lw      $16, 4($2)
        sw      $16, 0($2)
        sw      $15, 4($2)
        jr      $31
```

- Machine (object) code (for MIPS)

```
000000 00000 00101 0001000010000000
000000 00100 00010 0001000000100000
100011 00010 01111 0000000000000000
100011 00010 10000 0000000000000100
101011 00010 10000 0000000000000000
101011 00010 01111 0000000000000100
000000 11111 00000 0000000000001000
```



Below the Program

- High-level language program (in C)

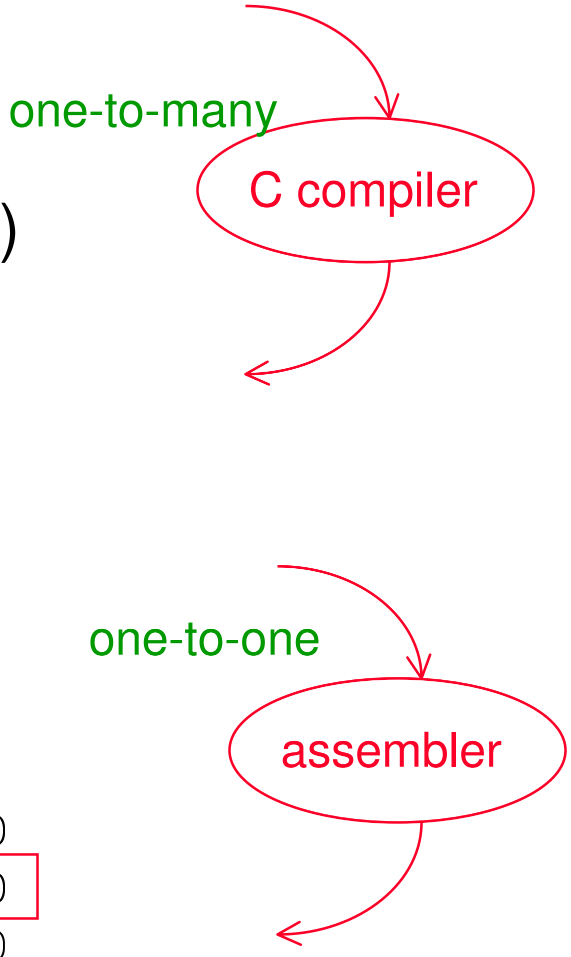
```
swap (int v[], int k)
. . .
```

- Assembly language program (for MIPS)

```
swap:   sll     $2, $5, 2
        add     $2, $4, $2
        lw      $15, 0($2)
        lw      $16, 4($2)
        sw      $16, 0($2)
        sw      $15, 4($2)
        jr      $31
```

- Machine (object) code (for MIPS)

```
000000 00000 00101 0001000010000000
000000 00100 00010 0001000000100000
100011 00010 01111 0000000000000000
100011 00010 10000 0000000000000100
101011 00010 10000 0000000000000000
101011 00010 01111 0000000000000100
000000 11111 00000 0000000000001000
```


one-to-many

C compiler

one-to-one

assembler

Below the Program

- High-level language program (in C)

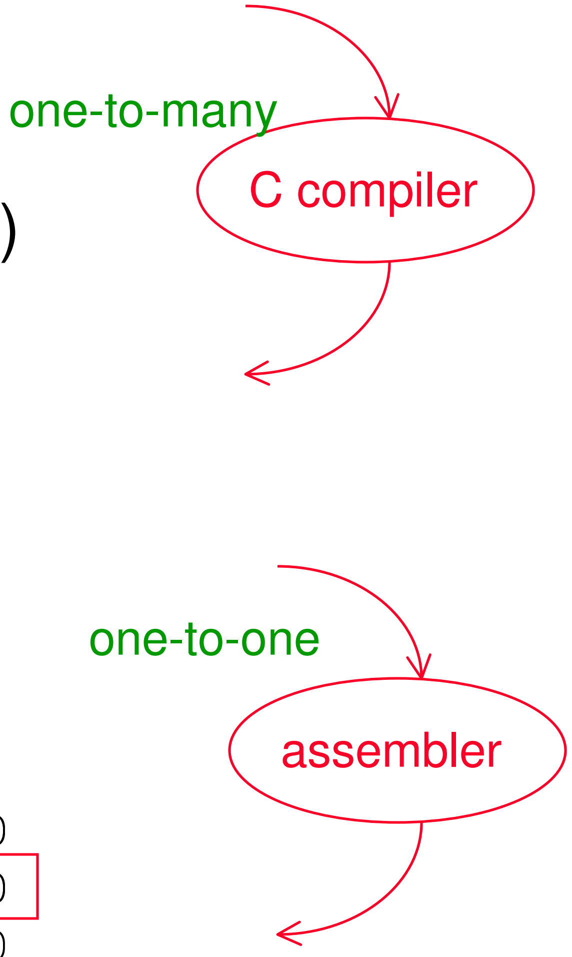
```
swap (int v[], int k)
. . .
```

- Assembly language program (for MIPS)

```
swap:   sll     $2, $5, 2
        add     $2, $4, $2
        lw      $15, 0($2)
        lw      $16, 4($2)
        sw      $16, 0($2)
        sw      $15, 4($2)
        jr      $31
```

- Machine (object) code (for MIPS)

```
000000 00000 00101 0001000010000000
000000 00100 00010 0001000000100000
100011 00010 01111 0000000000000000
100011 00010 10000 0000000000000100
101011 00010 10000 0000000000000000
101011 00010 01111 0000000000000100
000000 11111 00000 0000000000001000
```



Below the Program

- High-level language program (in C)

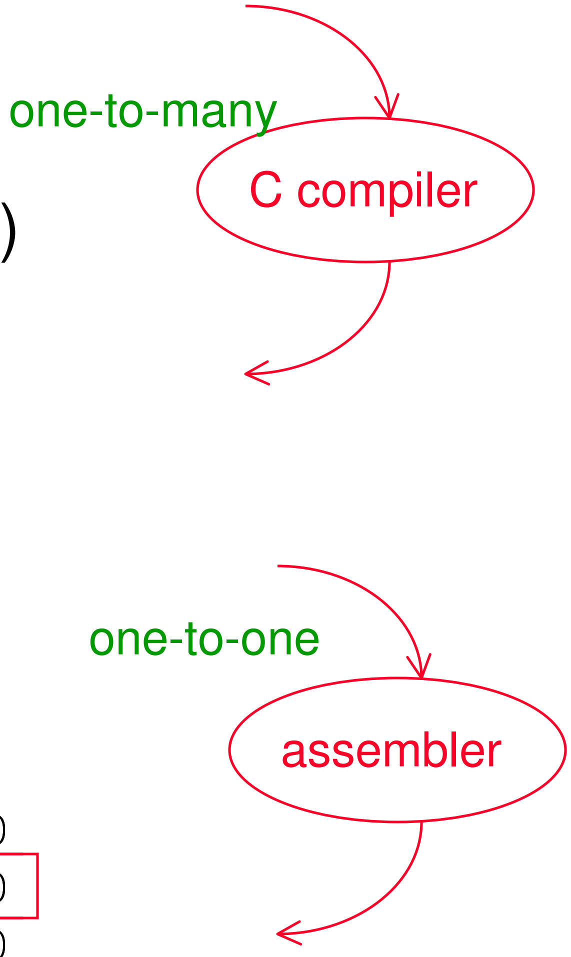
```
swap (int v[], int k)
. . .
```

- Assembly language program (for MIPS)

```
swap:   sll    $2, $5, 2
        add    $2, $4, $2
        lw     $15, 0($2)
        lw     $16, 4($2)
        sw     $16, 0($2)
        sw     $15, 4($2)
        jr     $31
```

- Machine (object) code (for MIPS)

```
000000 00000 00101 0001000010000000
000000 00100 00010 0001000000100000
100011 00010 01111 0000000000000000
100011 00010 10000 0000000000000100
101011 00010 10000 0000000000000000
101011 00010 01111 0000000000000100
000000 11111 00000 0000000000001000
```



Below the Program

- High-level language program (in C)

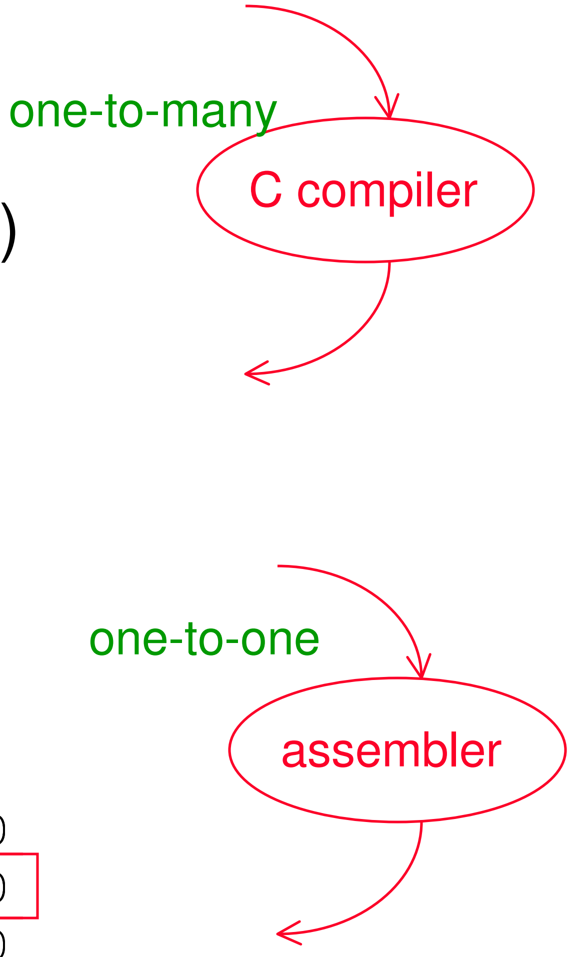
```
swap (int v[], int k)
. . .
```

- Assembly language program (for MIPS)

```
swap:   sll     $2, $5, 2
        add     $2, $4, $2
        lw      $15, 0($2)
        lw      $16, 4($2)
        sw      $16, 0($2)
        sw      $15, 4($2)
        jr      $31
```

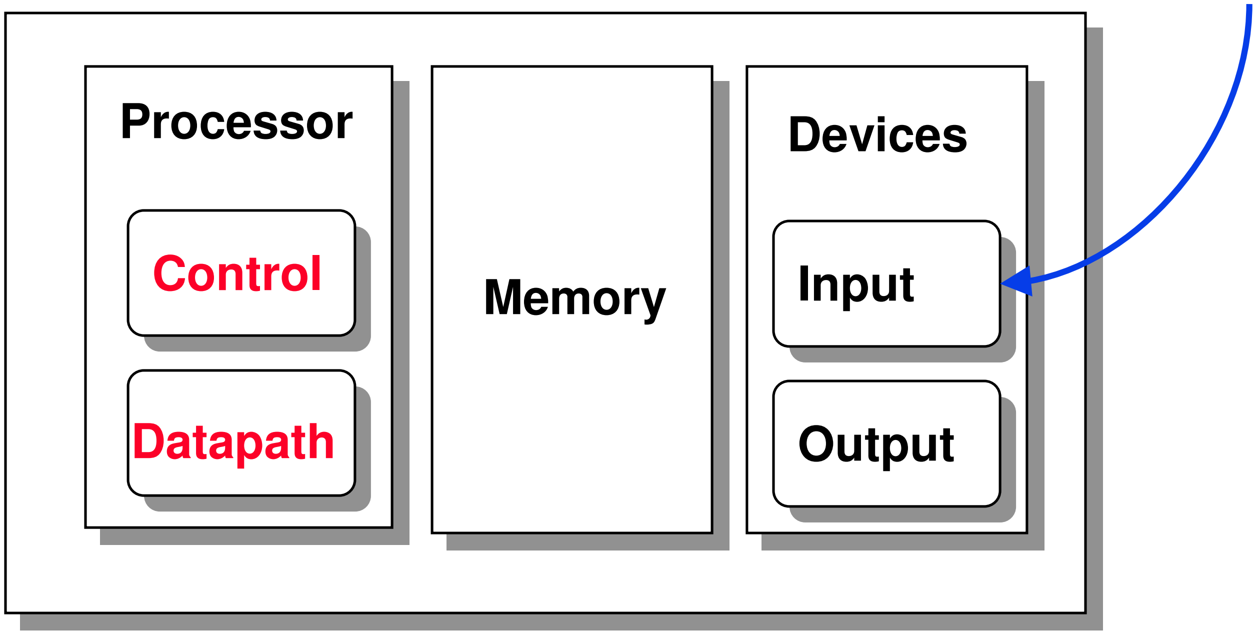
- Machine (object) code (for MIPS)

```
000000 00000 00101 0001000010000000
000000 00100 00010 0001000000100000
100011 00010 01111 0000000000000000
100011 00010 10000 0000000000000100
101011 00010 10000 0000000000000000
101011 00010 01111 0000000000000100
000000 11111 00000 0000000000001000
```

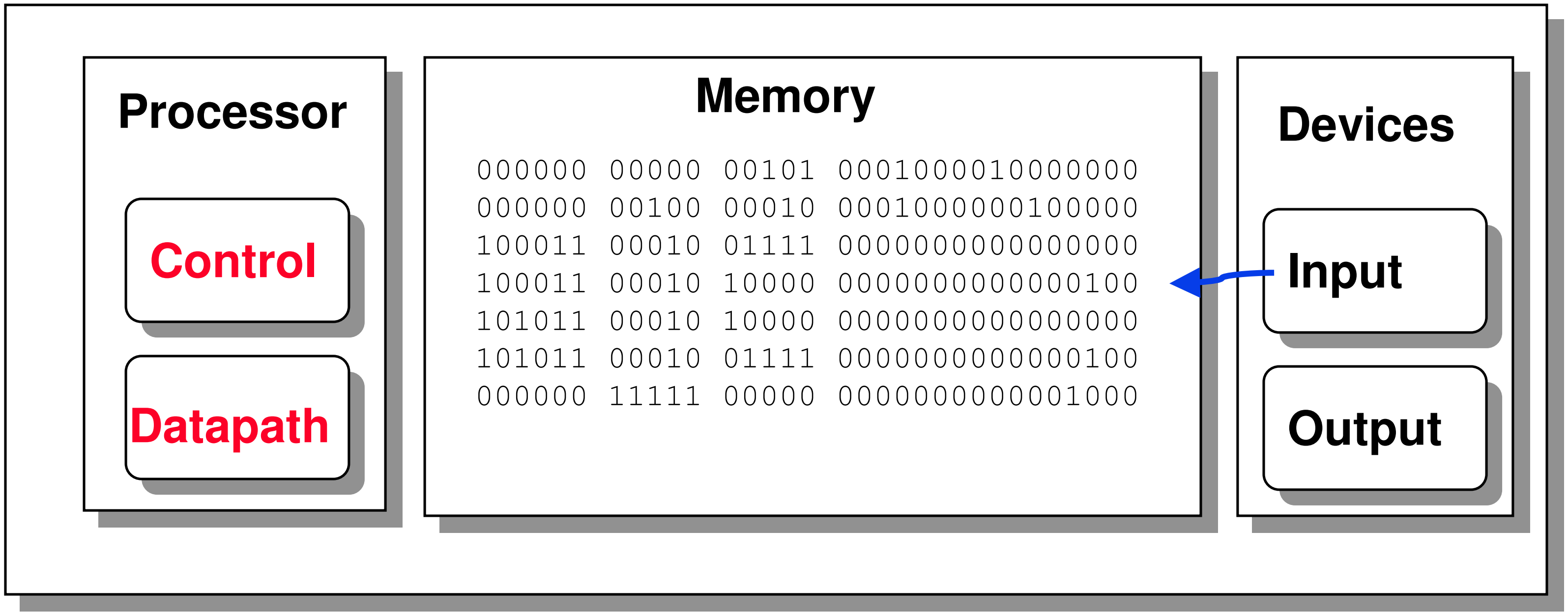


Input Device Inputs Object Code

```
000000 00000 00101 0001000010000000
000000 00100 00010 0001000000100000
100011 00010 01111 0000000000000000
100011 00010 10000 0000000000000100
101011 00010 10000 0000000000000000
101011 00010 01111 0000000000000100
000000 11111 00000 0000000000001000
```



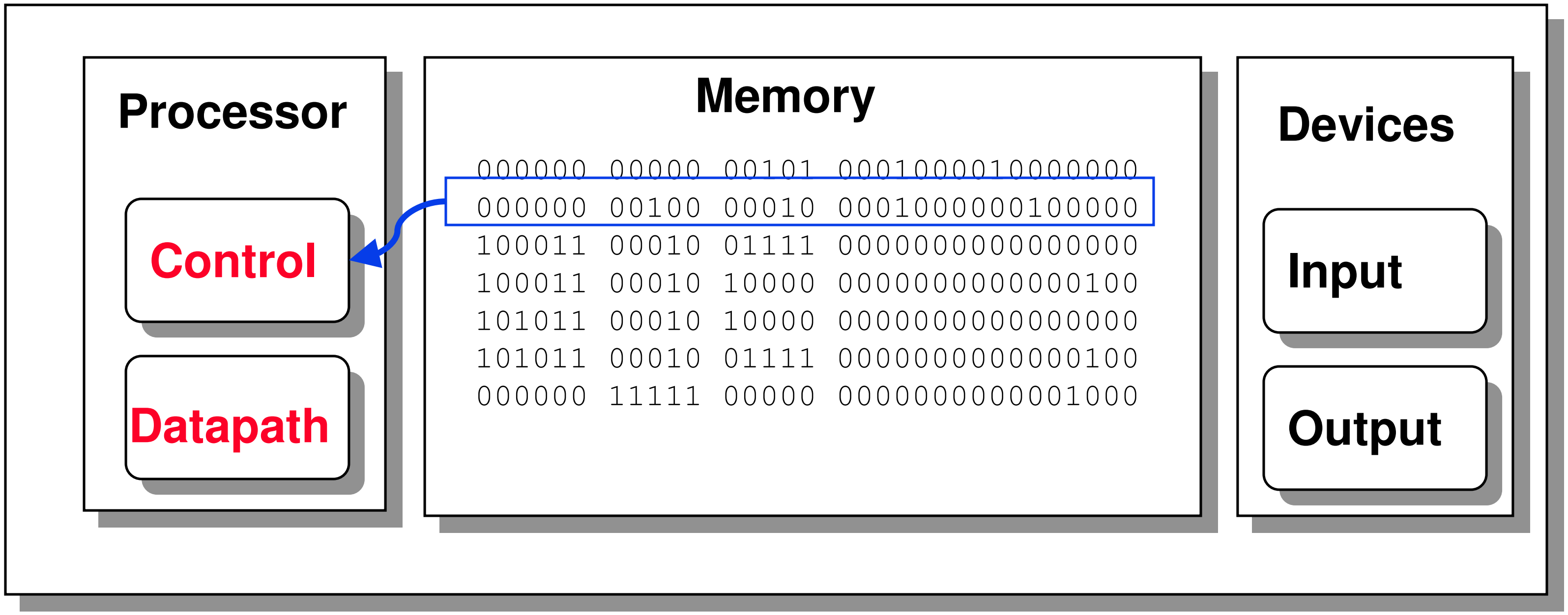
Object Code Stored in Memory



```
000000 00000 00101 0001000010000000
000000 00100 00010 0001000000100000
100011 00010 01111 0000000000000000
100011 00010 10000 0000000000000100
101011 00010 10000 0000000000000000
101011 00010 01111 0000000000000100
000000 11111 00000 0000000000001000
```

Processor Fetches an Instruction

Processor fetches an instruction from memory

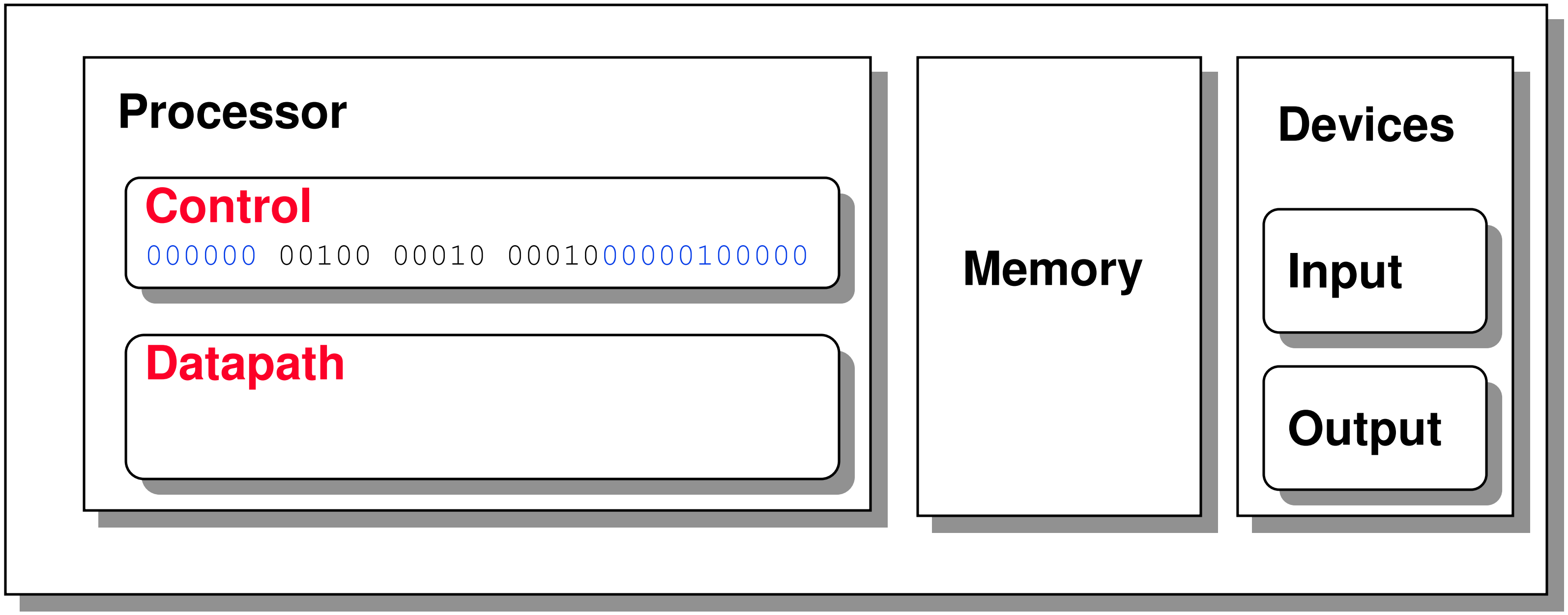


```
000000 00000 00101 0001000010000000
000000 00100 00010 0001000000100000
100011 00010 01111 0000000000000000
100011 00010 10000 0000000000000100
101011 00010 10000 0000000000000000
101011 00010 01111 0000000000000100
000000 11111 00000 0000000000001000
```

Where does it fetch from?

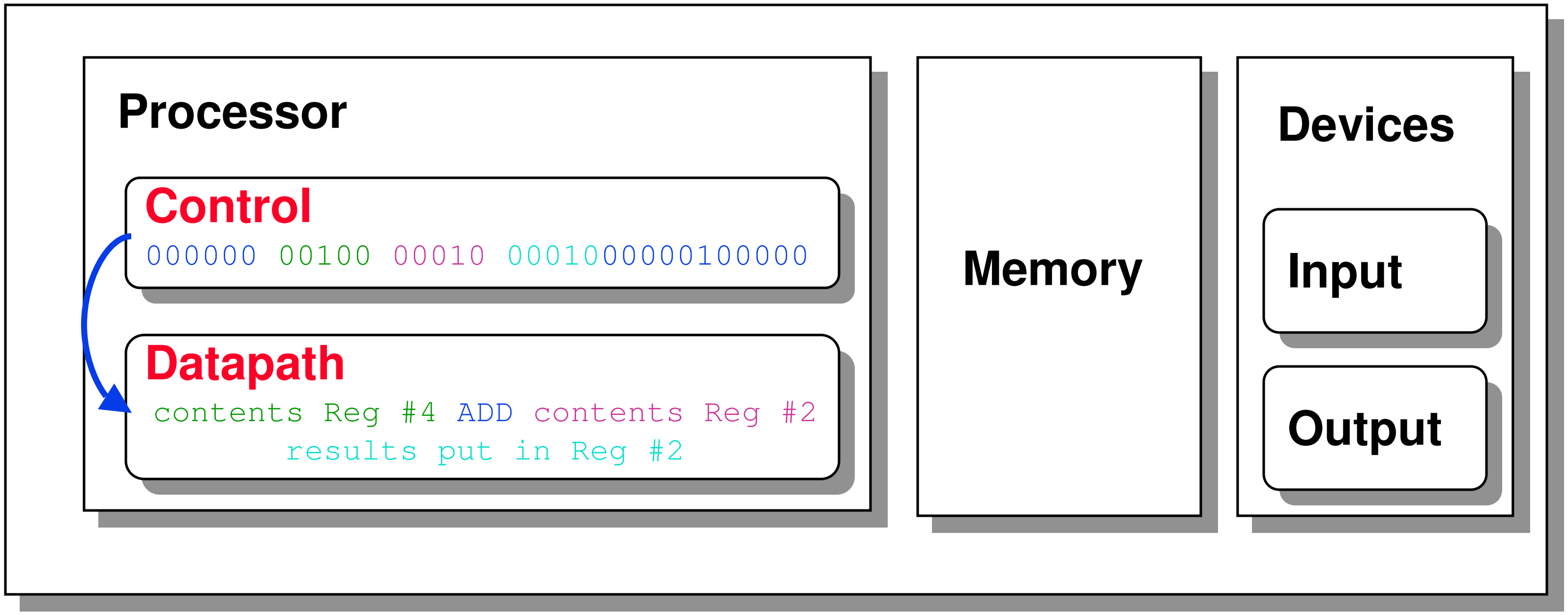
Control Decodes the Instruction

Control decodes the instruction to determine what to execute



Datapath Executes the Instruction

Datapath executes the instruction as directed by control



Processor

Control

000000 00100 00010 0001000000100000

Datapath

contents Reg #4 ADD contents Reg #2
results put in Reg #2

Memory

Devices

Input

Output

Processor Organization

Processor Organization

- **Control** needs to have the
  - Ability to input instructions from memory
  - Logic and means to control instruction sequencing
  - Logic and means to issue signals that control the way information flows between datapath components
  - Logic and means to control what operations the datapath's functional units perform

Processor Organization

- **Control** needs to have the
  - Ability to input instructions from memory
  - Logic and means to control instruction sequencing
  - Logic and means to issue signals that control the way information flows between datapath components
  - Logic and means to control what operations the datapath's functional units perform
- **Datapath** needs to have the
  - Components - functional units (e.g., adder) and storage locations (e.g., register file) - needed to execute instructions
  - Components interconnected so that the instructions can be accomplished
  - Ability to load data from and store data to memory

Processor Organization

❑ **Control** needs to have the

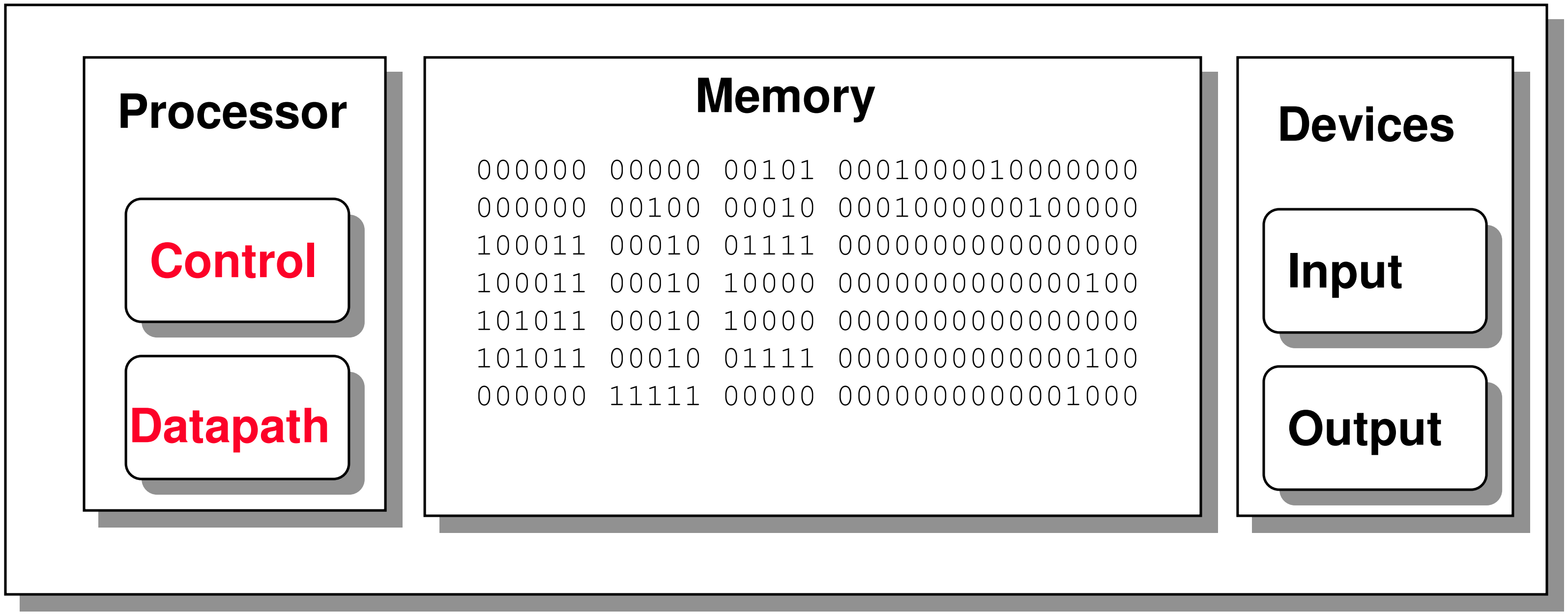
- Ability to input instructions from memory
- Logic and means to control instruction sequencing
- Logic and means to issue signals that control the way information flows between datapath components
- Logic and means to control what operations the datapath's functional units perform

❑ **Datapath** needs to have the

- Components - functional units (e.g., adder) and storage locations (e.g., register file) - needed to execute instructions
- Components interconnected so that the instructions can be accomplished
- Ability to load data from and store data to memory

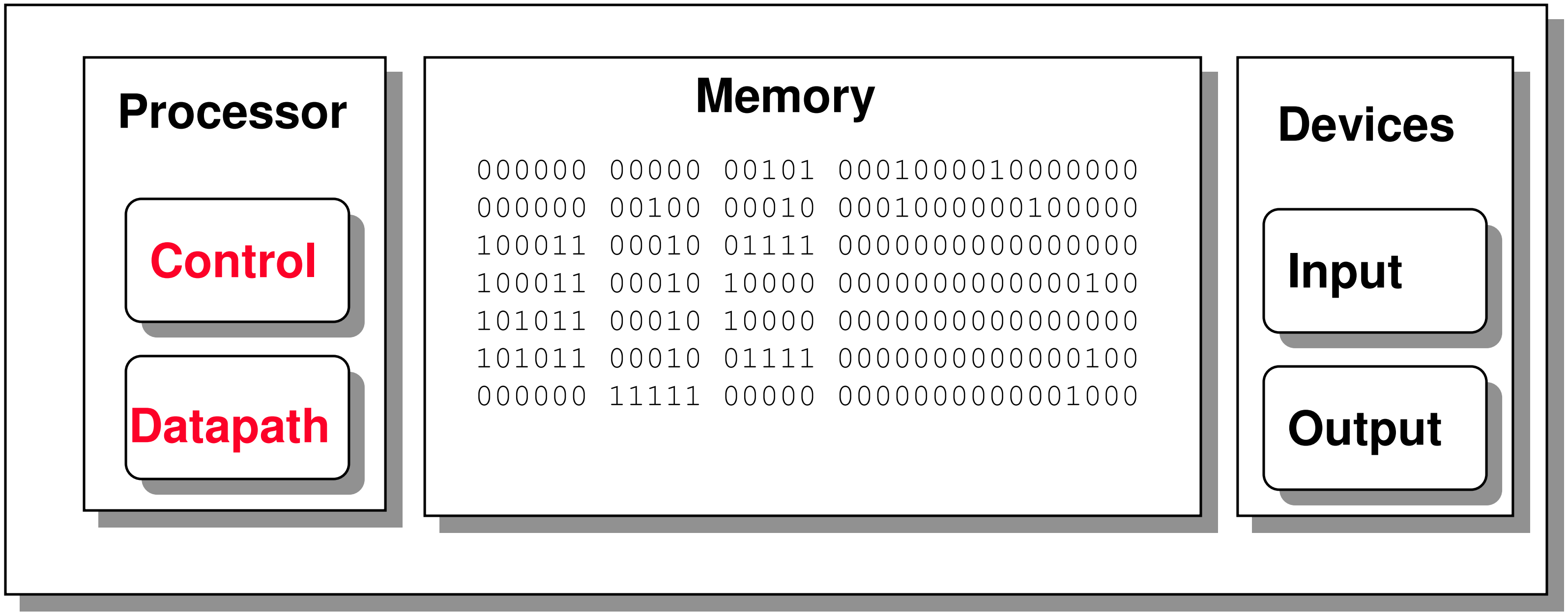
Where does it load and store from and to?

What Happens Next?



```
000000 00000 00101 0001000010000000
000000 00100 00010 0001000000100000
100011 00010 01111 0000000000000000
100011 00010 10000 0000000000000100
101011 00010 10000 0000000000000000
101011 00010 01111 0000000000000100
000000 11111 00000 0000000000001000
```

What Happens Next?



Processor

Control

Datapath

Memory

```
000000 00000 00101 0001000010000000
000000 00100 00010 0001000000100000
100011 00010 01111 0000000000000000
100011 00010 10000 0000000000000100
101011 00010 10000 0000000000000000
101011 00010 01111 0000000000000100
000000 11111 00000 0000000000001000
```

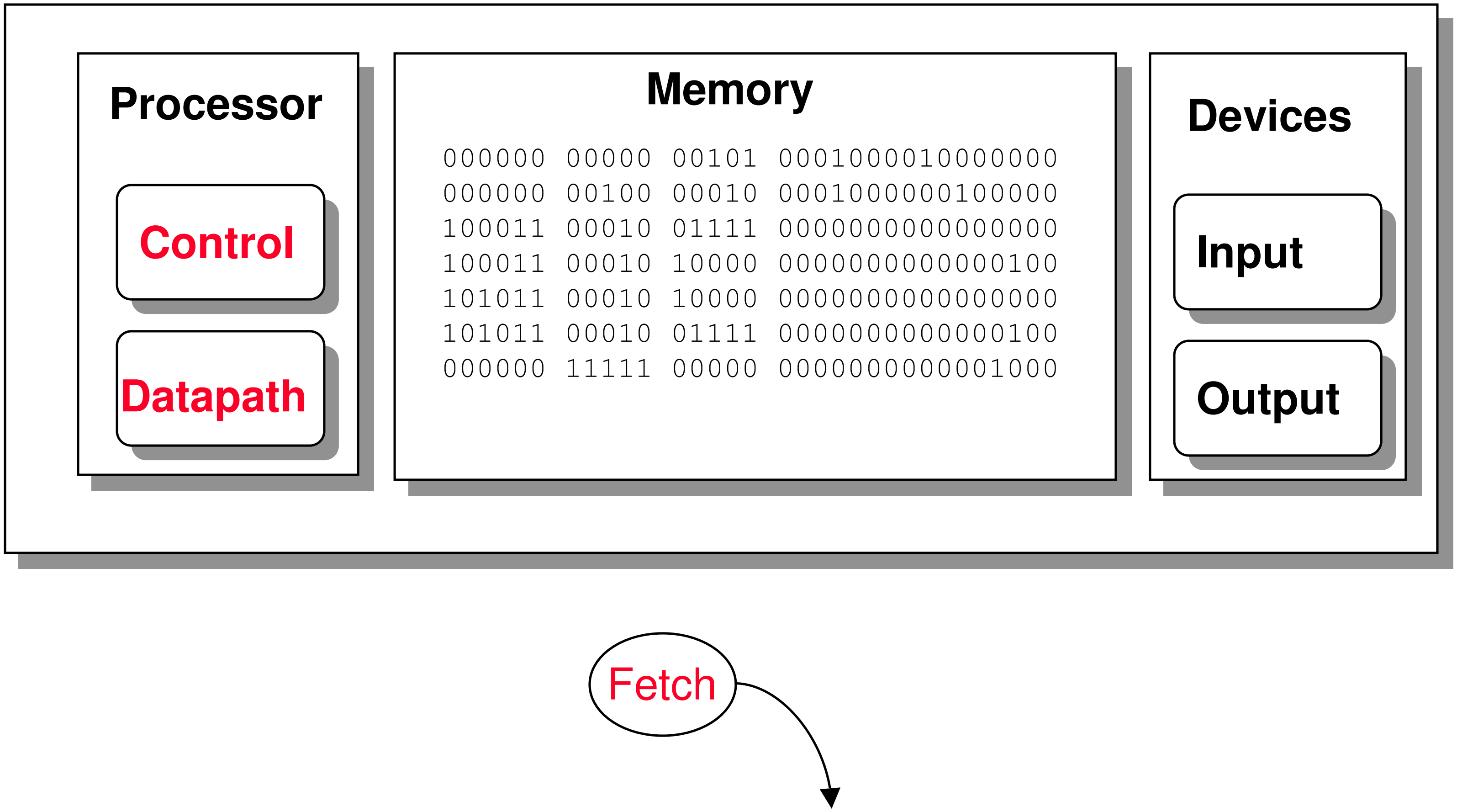
Devices

Input

Output

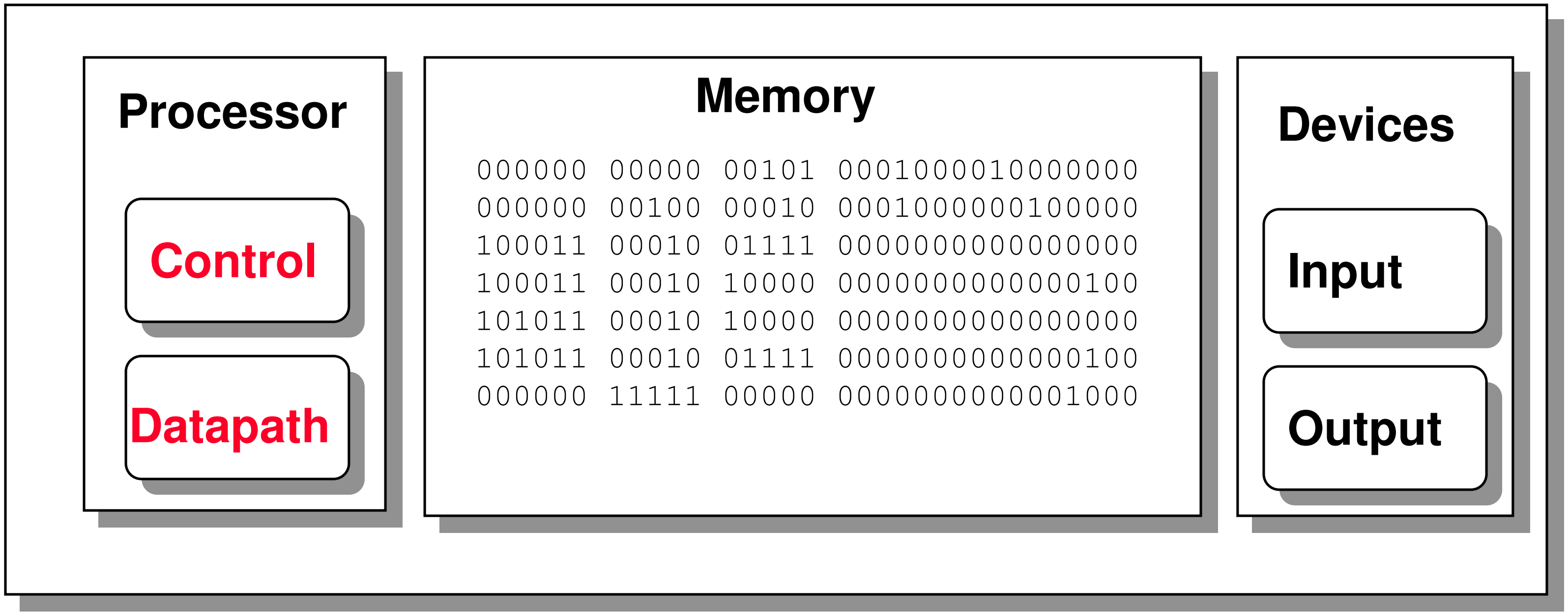
Fetch

What Happens Next?

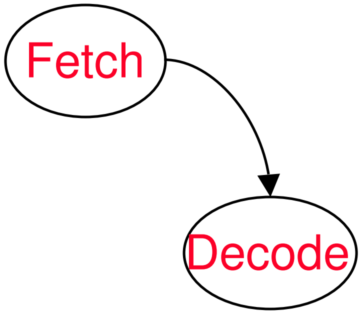


```
000000 00000 00101 0001000010000000
000000 00100 00010 0001000000100000
100011 00010 01111 0000000000000000
100011 00010 10000 0000000000000100
101011 00010 10000 0000000000000000
101011 00010 01111 0000000000000100
000000 11111 00000 0000000000001000
```

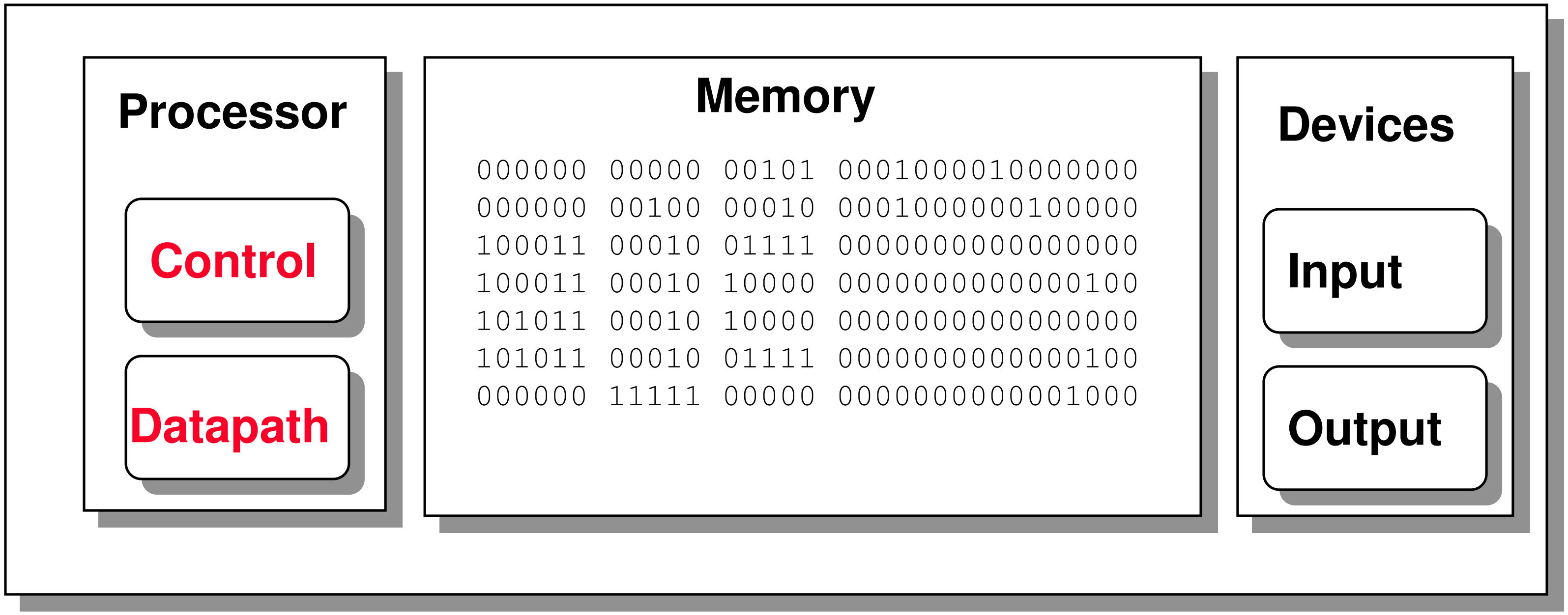
What Happens Next?



```
000000 00000 00101 0001000010000000
000000 00100 00010 0001000000100000
100011 00010 01111 0000000000000000
100011 00010 10000 0000000000000100
101011 00010 10000 0000000000000000
101011 00010 01111 0000000000000100
000000 11111 00000 0000000000001000
```



What Happens Next?



Processor

Control

Datapath

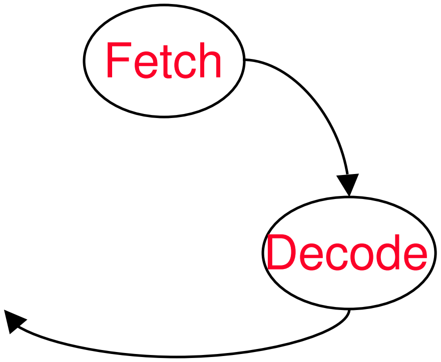
Memory

```
000000 00000 00101 0001000010000000
000000 00100 00010 0001000000100000
100011 00010 01111 0000000000000000
100011 00010 10000 0000000000000100
101011 00010 10000 0000000000000000
101011 00010 01111 0000000000000100
000000 11111 00000 0000000000001000
```

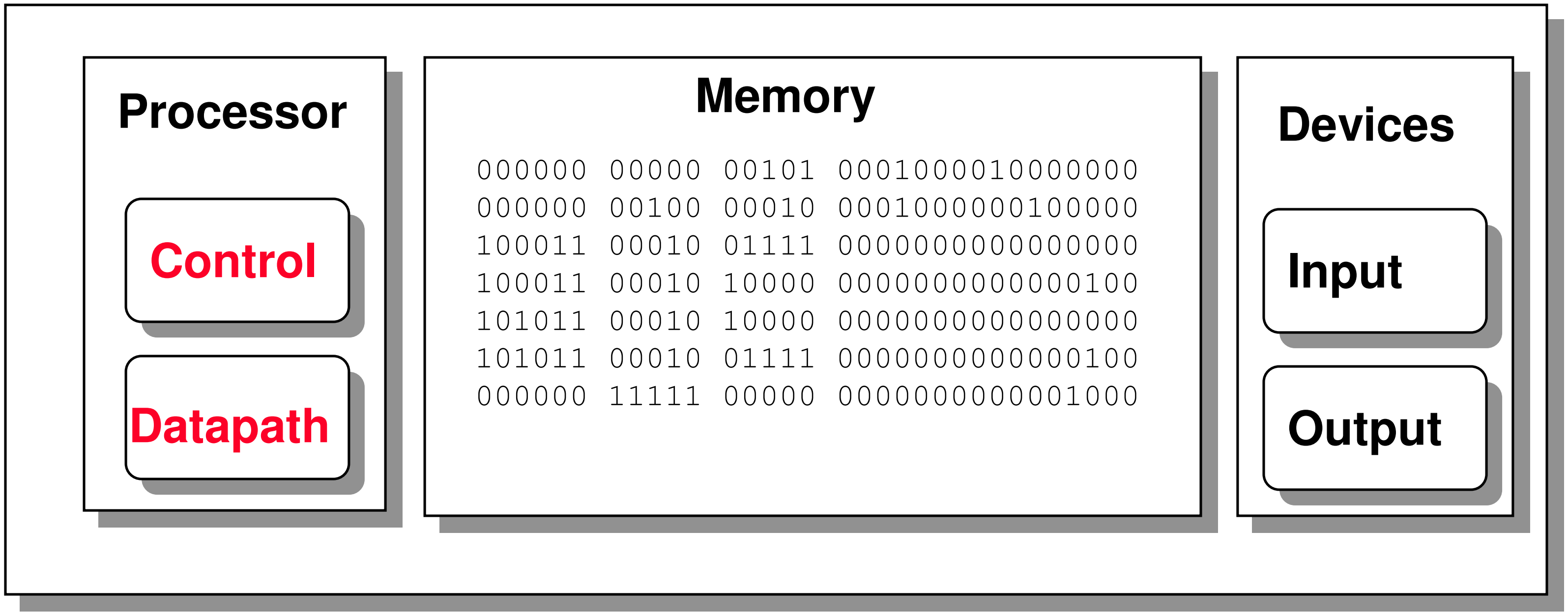
Devices

Input

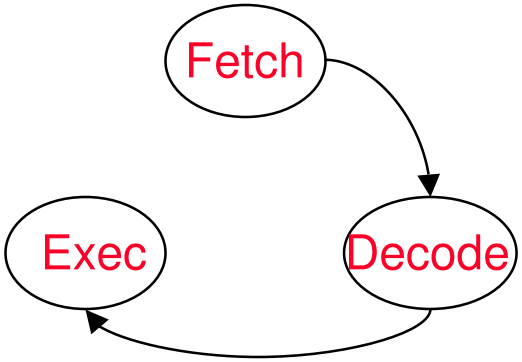
Output



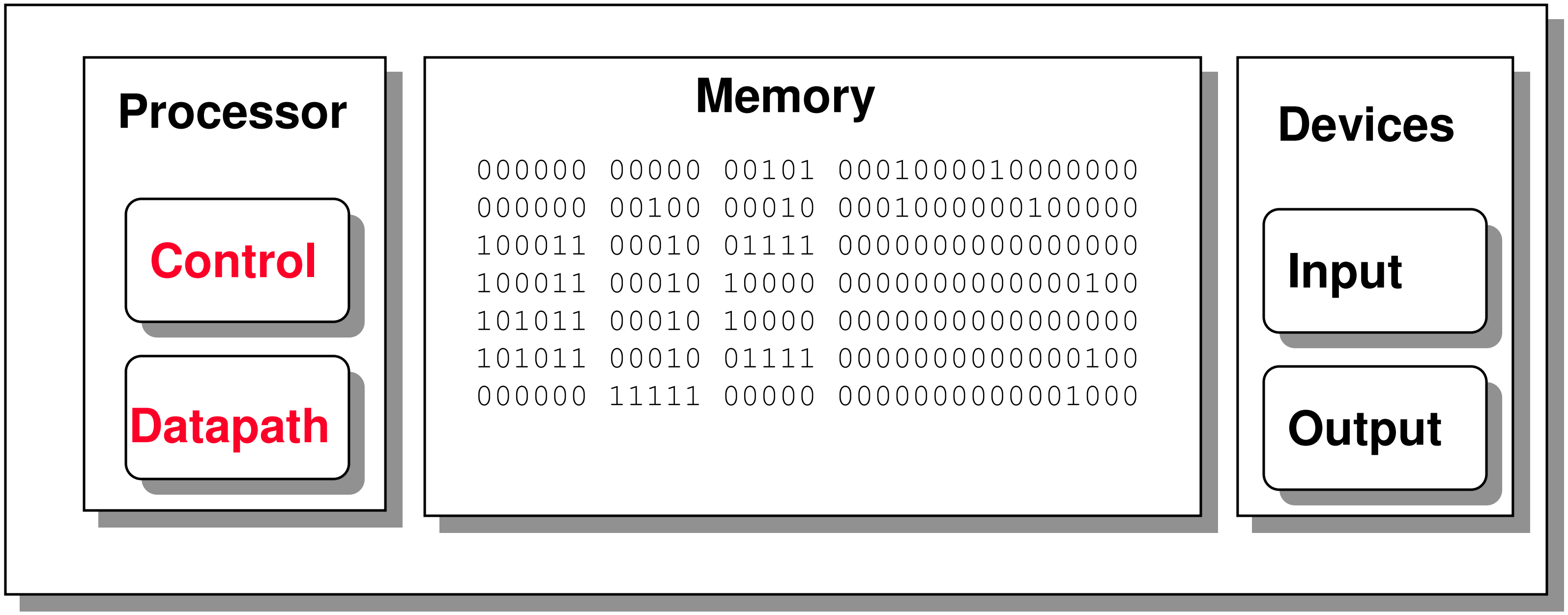
What Happens Next?



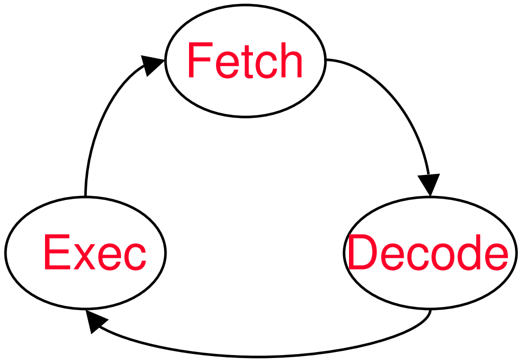
```
000000 00000 00101 0001000010000000
000000 00100 00010 0001000000100000
100011 00010 01111 0000000000000000
100011 00010 10000 0000000000000100
101011 00010 10000 0000000000000000
101011 00010 01111 0000000000000100
000000 11111 00000 0000000000001000
```



What Happens Next?

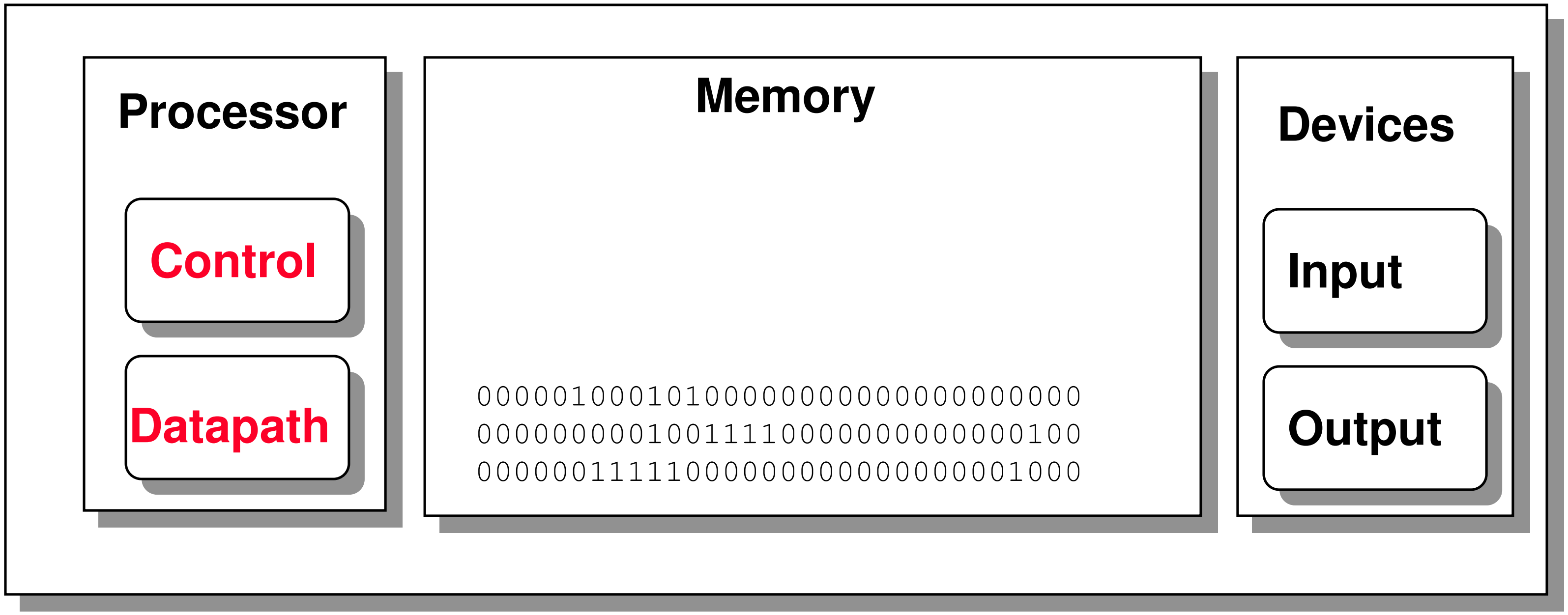


```
000000 00000 00101 0001000010000000
000000 00100 00010 0001000000100000
100011 00010 01111 0000000000000000
100011 00010 10000 0000000000000100
101011 00010 10000 0000000000000000
101011 00010 01111 0000000000000100
000000 11111 00000 0000000000001000
```



Output Data Stored in Memory

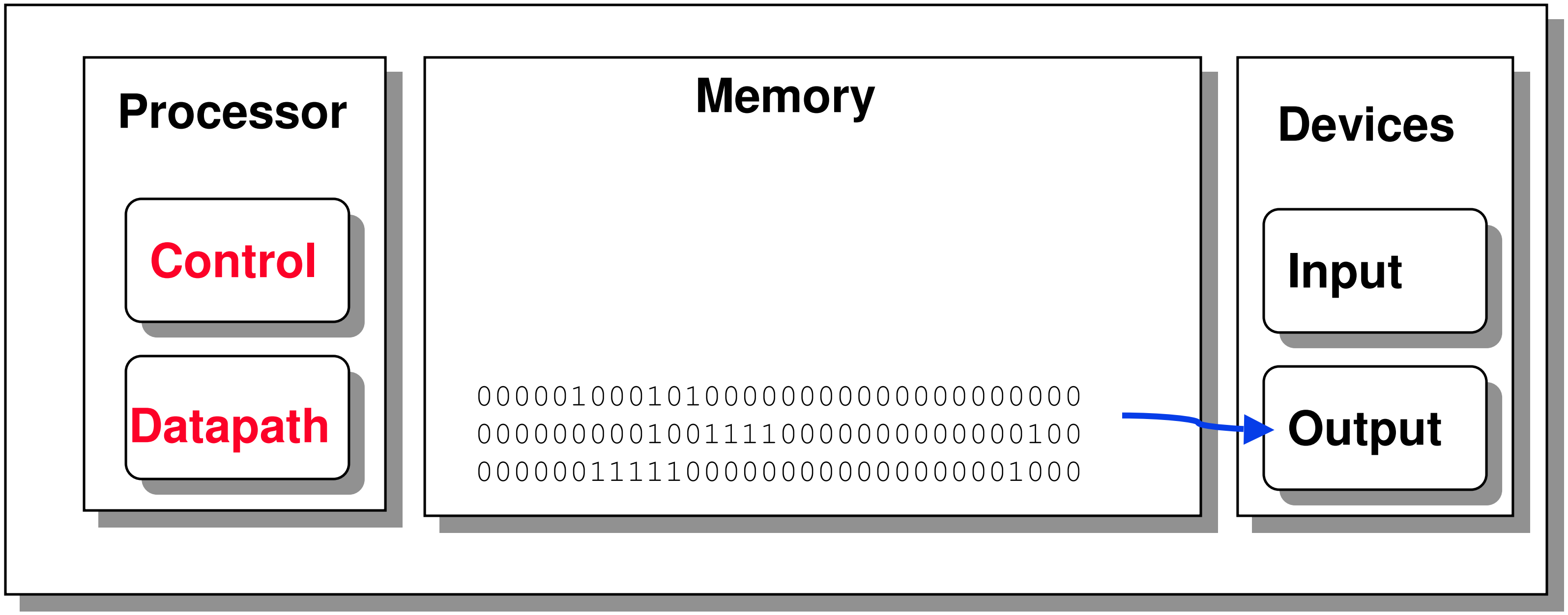
At program completion the data to be output resides in memory



```
00000100010100000000000000000000
00000000010011110000000000000100
00000011111000000000000000001000
```

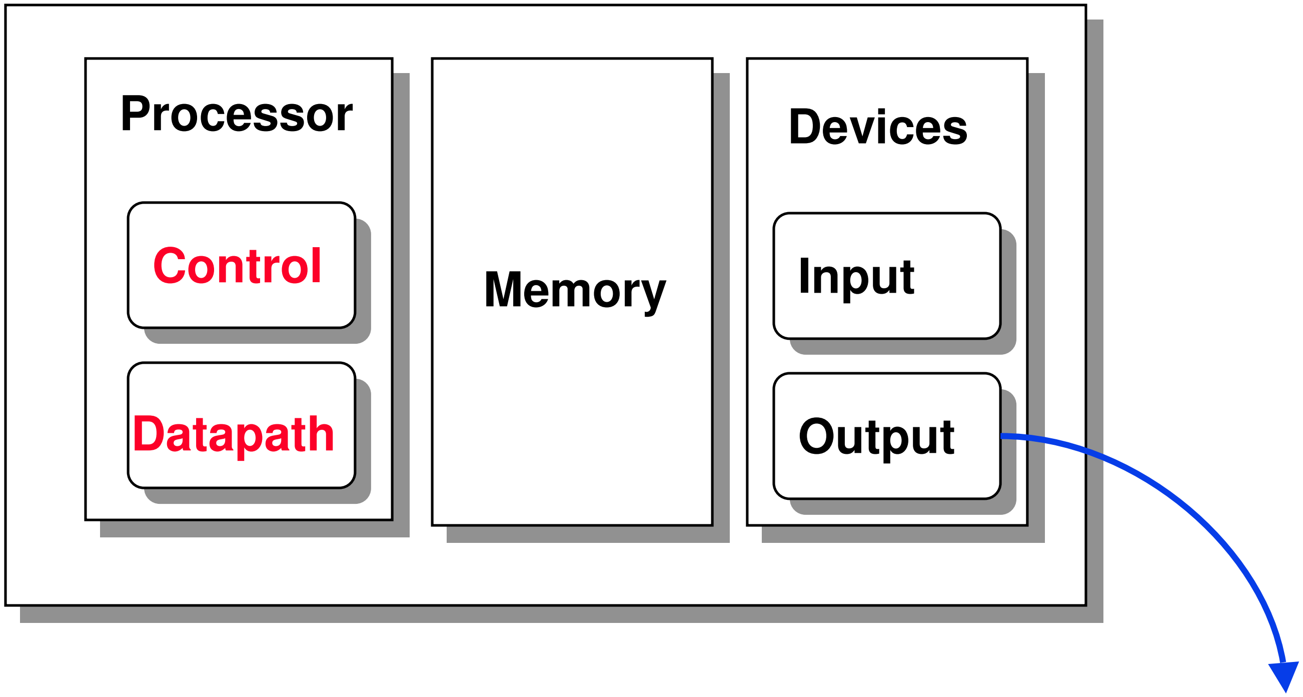
Output Data Stored in Memory

At program completion the data to be output
resides in memory



```
00000100010100000000000000000000
00000000010011110000000000000100
00000011111000000000000000001000
```

Output Device Outputs Data

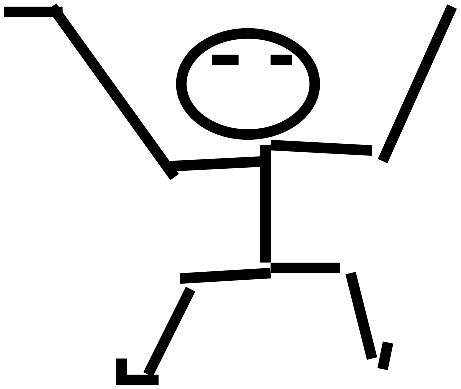


```
00000100010100000000000000000000
00000000010011110000000000000100
00000011111000000000000000001000
```

The Instruction Set Architecture

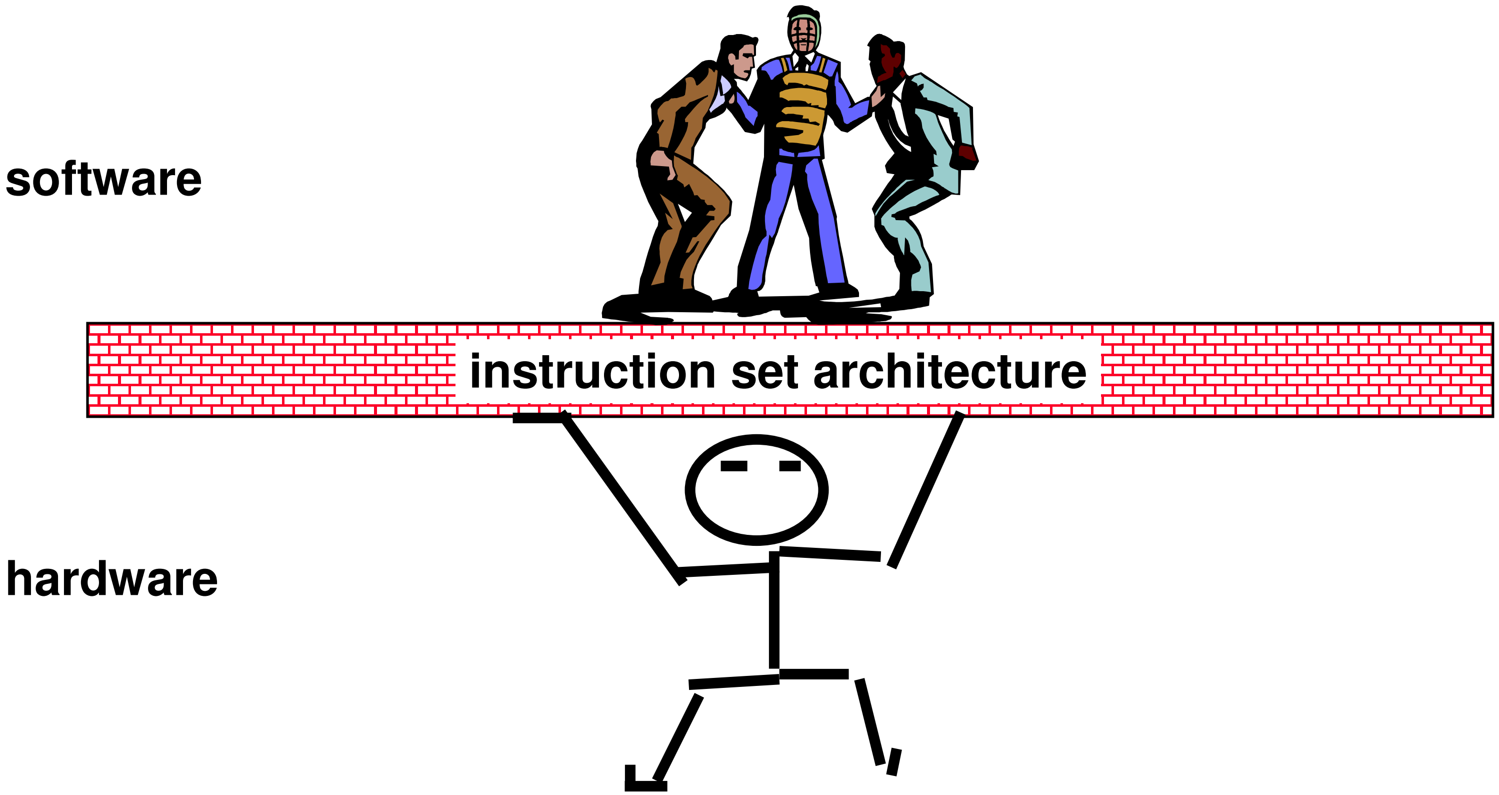
software

instruction set architecture



hardware

The Instruction Set Architecture

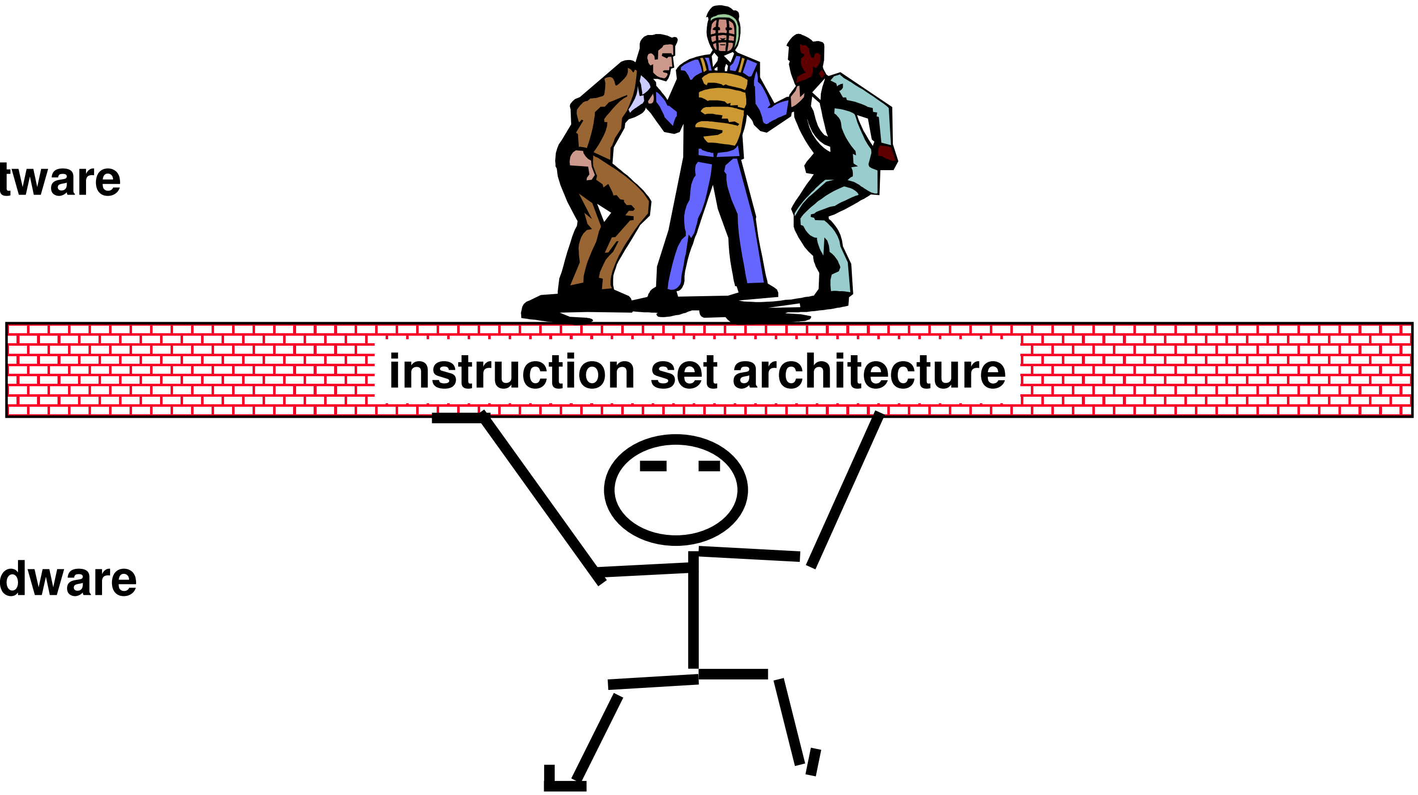


software

instruction set architecture

hardware

The Instruction Set Architecture



software

instruction set architecture

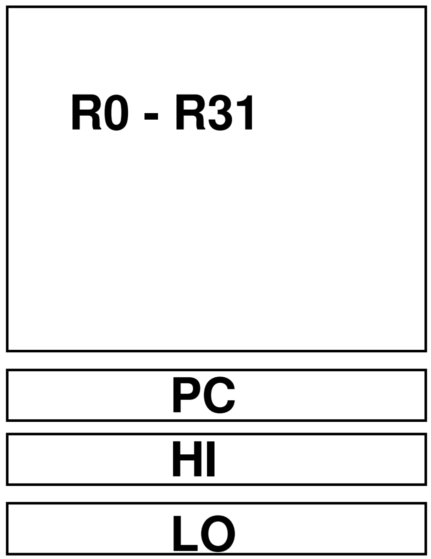
hardware

The interface description separating the software and hardware.

MIPS R3000 Instruction Set Architecture

- Instruction Categories
  - Load/Store
  - Computational
  - Jump and Branch
  - Floating Point
    - coprocessor
  - Memory Management
  - Special

Registers



R0 - R31

PC

HI

LO

- 3 Instruction Formats: all 32 bits wide

OP	rs	rt	rd	sa	funct

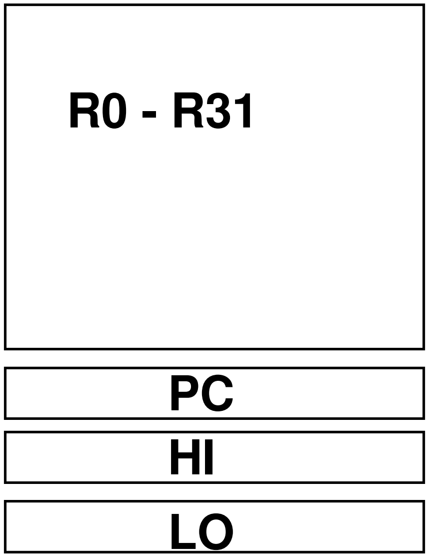
OP	rs	rt	immediate

OP	jump target

MIPS R3000 Instruction Set Architecture

- Instruction Categories
  - Load/Store
  - Computational
  - Jump and Branch
  - Floating Point
    - coprocessor
  - Memory Management
  - Special

Registers



- **3 Instruction Formats: all 32 bits wide**

<table>
<tr><td><b>OP</b></td><td><b>rs</b></td><td><b>rt</b></td><td><b>rd</b></td><td><b>sa</b></td><td><b>funct</b></td></tr>
<tr><td><b>OP</b></td><td><b>rs</b></td><td><b>rt</b></td><td colspan="3"><b>immediate</b></td></tr>
<tr><td><b>OP</b></td><td colspan="5"><b>jump target</b></td></tr>
</table>

Q: How many already familiar with MIPS ISA?

Assembly Language Instructions

- Language of the machine
- More primitive than higher level languages

  e.g., no sophisticated control flow
- Very restrictive

  e.g., MIPS arithmetic instructions
- We'll be working with the MIPS instruction set architecture
  - similar to other architectures developed since the 1980's
  - used by NEC, Nintendo, Silicon Graphics, Sony, …

Design goals: maximize performance, minimize cost, reduce design time, minimize memory space (embedded systems), minimize power consumption (mobile systems)

RISC - Reduced Instruction Set Computer

- RISC philosophy
  - fixed instruction lengths
  - load-store instruction sets
  - limited addressing modes
  - limited operations
- MIPS, Sun SPARC, HP PA-RISC, IBM PowerPC, Intel (Compaq) Alpha, …
- Instruction sets are measured by how well compilers use them as opposed to how well assembly language programmers use them

MIPS Arithmetic Instruction

- MIPS assembly language arithmetic statement

```
add  $t0, $s1, $s2

sub  $t0, $s1, $s2
```

- Each arithmetic instruction performs only one operation
- Each arithmetic instruction specifies exactly three operands

  destination ← source1   op   source2

- Those operands are contained in the datapath's register file (`$t0, $s1,$s2`)
- Operand order is fixed (destination first)

MIPS Arithmetic Instruction

- MIPS assembly language arithmetic statement

```
add  $t0, $s1, $s2

sub  $t0, $s1, $s2
```

- Each arithmetic instruction performs only one operation

- Each arithmetic instruction specifies exactly three operands

  destination ← source1 op source2

- Those operands are contained in the datapath's register file (`$t0, $s1,$s2`)

- Operand order is fixed (destination first)

MIPS Arithmetic Instruction

- MIPS assembly language arithmetic statement

```
add   $t0, $s1, $s2

sub   $t0, $s1, $s2
```

- Each arithmetic instruction performs only one operation
- Each arithmetic instruction specifies exactly three operands

destination ← source1   op   source2

- Those operands are contained in the datapath's register file (`$t0, $s1,$s2`)
- Operand order is fixed (destination first)

MIPS Arithmetic Instruction

- MIPS assembly language arithmetic statement

```
add  $t0, $s1, $s2

sub  $t0, $s1, $s2
```

- Each arithmetic instruction performs only one operation
- Each arithmetic instruction specifies exactly three operands

destination ← source1 op source2

- Those operands are contained in the datapath's register file (`$t0, $s1,$s2`)
- Operand order is fixed (destination first)

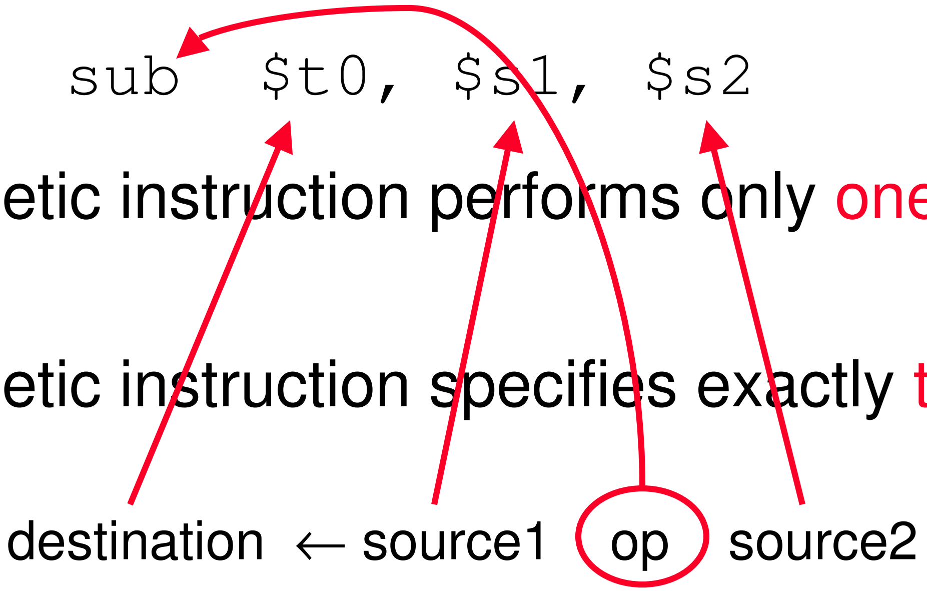
MIPS Arithmetic Instruction

- MIPS assembly language arithmetic statement

```
add  $t0, $s1, $s2
sub  $t0, $s1, $s2
```

- Each arithmetic instruction performs only one operation

- Each arithmetic instruction specifies exactly three operands



destination ← source1 op source2

- Those operands are contained in the datapath's register file (`$t0, $s1,$s2`)

- Operand order is fixed (destination first)

Compiling More Complex Statements

- Assuming variable b is stored in register `$s1`, c is stored in `$s2`, and d is stored in `$s3` and the result is to be left in `$s0`, what is the assembler equivalent to the C statement

```
h = (b - c) + d
```

Compiling More Complex Statements

- Assuming variable b is stored in register `$s1`, c is stored in `$s2`, and d is stored in `$s3` and the result is to be left in `$s0`, what is the assembler equivalent to the C statement

```
h = (b - c) + d
```

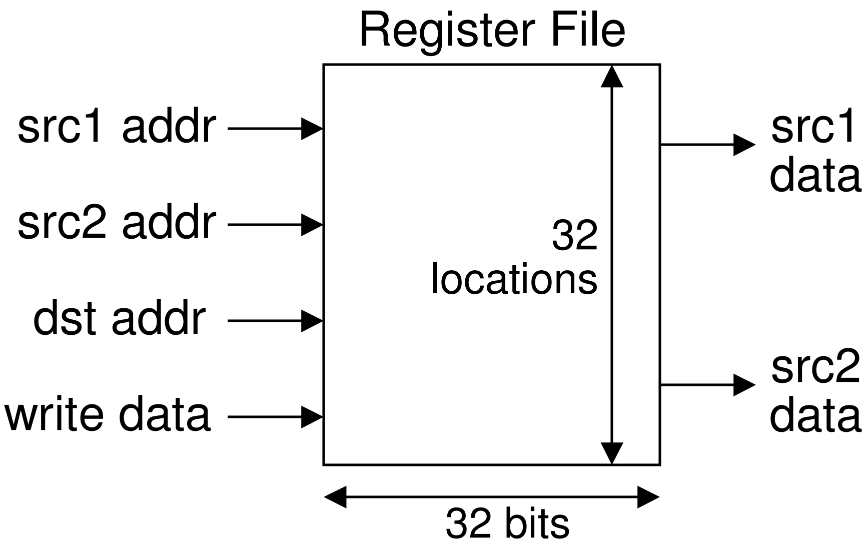
```
sub  $t0, $s1, $s2
add  $s0, $t0, $s3
```

MIPS Register File

- Operands of arithmetic instructions must be from a limited number of special locations contained in the datapath's register file
  - Holds thirty-two 32-bit registers
    - With two read ports and
    - One write port
- Registers are
  - Faster than main memory
  - Easier for a compiler to use
    - e.g., (A*B) – (C*D) – (E*F) can do multiplies in any order vs. stack
  - Can hold variables so that
    - code density improves (since register are named with fewer bits than a memory location)
- Register addresses are indicated by using $

MIPS Register File

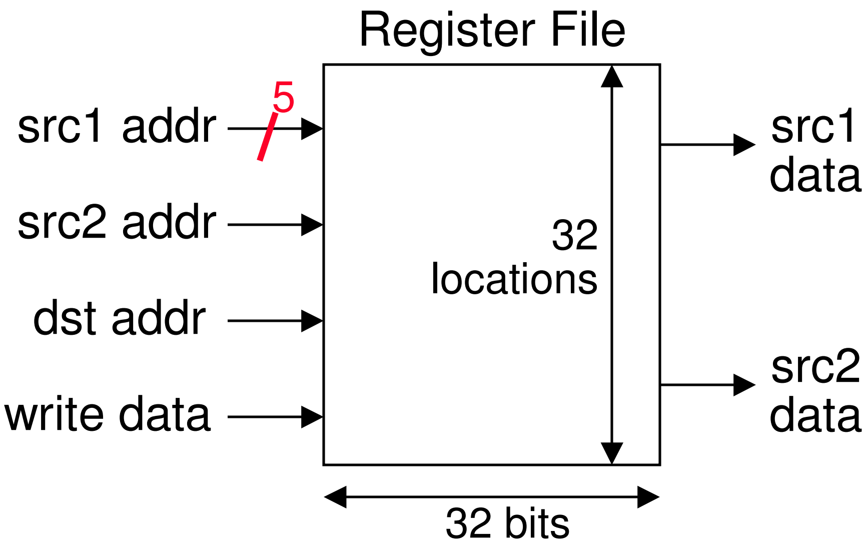
- Operands of arithmetic instructions must be from a limited number of special locations contained in the datapath’s register file
  - Holds thirty-two 32-bit registers
    - With two read ports and
    - One write port



- Registers are
  - Faster than main memory
  - Easier for a compiler to use
    - e.g., (A*B) – (C*D) – (E*F) can do multiplies in any order vs. stack
  - Can hold variables so that
    - code density improves (since register are named with fewer bits than a memory location)
- Register addresses are indicated by using $

MIPS Register File

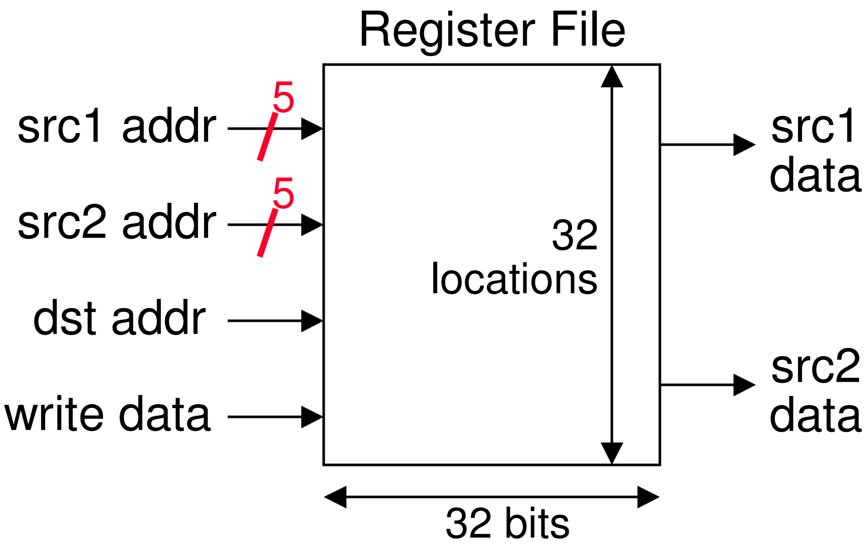
- Operands of arithmetic instructions must be from a limited number of special locations contained in the datapath's register file
  - Holds thirty-two 32-bit registers
    - With two read ports and
    - One write port



- Registers are
  - Faster than main memory
  - Easier for a compiler to use
    - e.g., (A*B) – (C*D) – (E*F) can do multiplies in any order vs. stack
  - Can hold variables so that
    - code density improves (since register are named with fewer bits than a memory location)
- Register addresses are indicated by using $

MIPS Register File

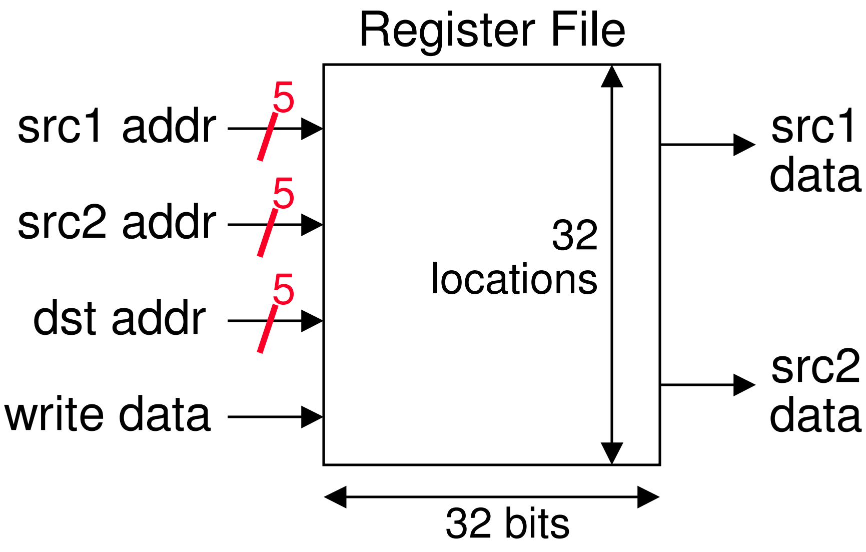
- Operands of arithmetic instructions must be from a limited number of special locations contained in the datapath's register file
  - Holds thirty-two 32-bit registers
    - With two read ports and
    - One write port



- Registers are
  - Faster than main memory
  - Easier for a compiler to use
    - e.g., (A*B) – (C*D) – (E*F) can do multiplies in any order vs. stack
  - Can hold variables so that
    - code density improves (since register are named with fewer bits than a memory location)
- Register addresses are indicated by using $

MIPS Register File

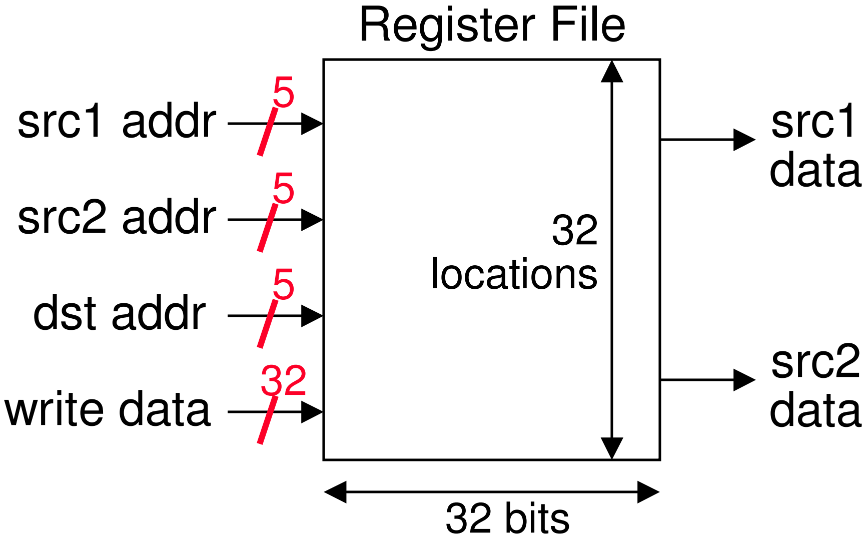
- Operands of arithmetic instructions must be from a limited number of special locations contained in the datapath's register file
  - Holds thirty-two 32-bit registers
    - With two read ports and
    - One write port



- Registers are
  - Faster than main memory
  - Easier for a compiler to use
    - e.g., (A*B) – (C*D) – (E*F) can do multiplies in any order vs. stack
  - Can hold variables so that
    - code density improves (since register are named with fewer bits than a memory location)
- Register addresses are indicated by using $

MIPS Register File

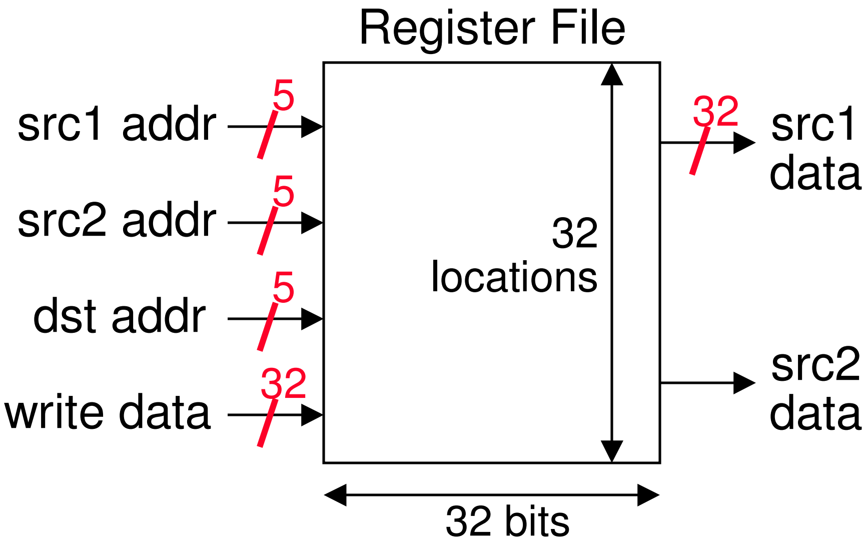
- Operands of arithmetic instructions must be from a limited number of special locations contained in the datapath's register file
  - Holds thirty-two 32-bit registers
    - With two read ports and
    - One write port



- Registers are
  - Faster than main memory
  - Easier for a compiler to use
    - e.g., (A*B) – (C*D) – (E*F) can do multiplies in any order vs. stack
  - Can hold variables so that
    - code density improves (since register are named with fewer bits than a memory location)
- Register addresses are indicated by using $

MIPS Register File

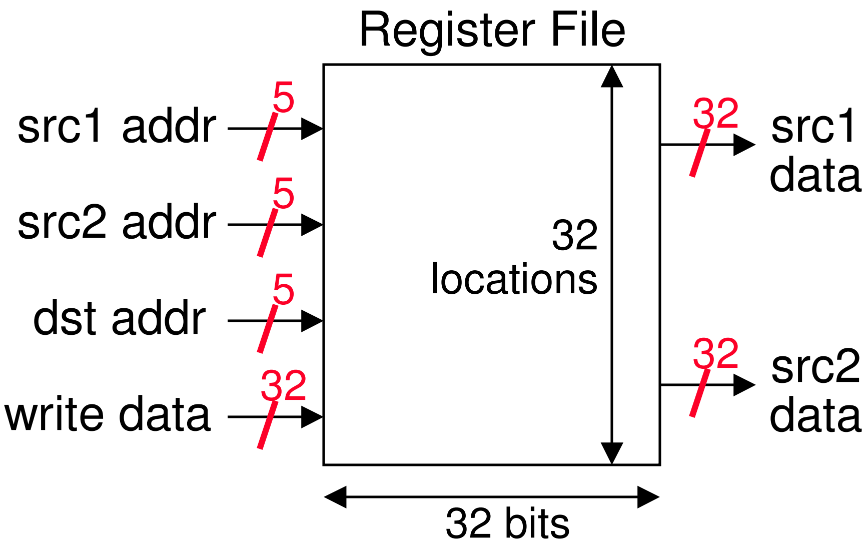
- Operands of arithmetic instructions must be from a limited number of special locations contained in the datapath's register file
  - Holds thirty-two 32-bit registers
    - With two read ports and
    - One write port



- Registers are
  - Faster than main memory
  - Easier for a compiler to use
    - e.g., (A*B) – (C*D) – (E*F) can do multiplies in any order vs. stack
  - Can hold variables so that
    - code density improves (since register are named with fewer bits than a memory location)
- Register addresses are indicated by using $

MIPS Register File

- Operands of arithmetic instructions must be from a limited number of special locations contained in the datapath's register file
  - Holds thirty-two 32-bit registers
    - With two read ports and
    - One write port



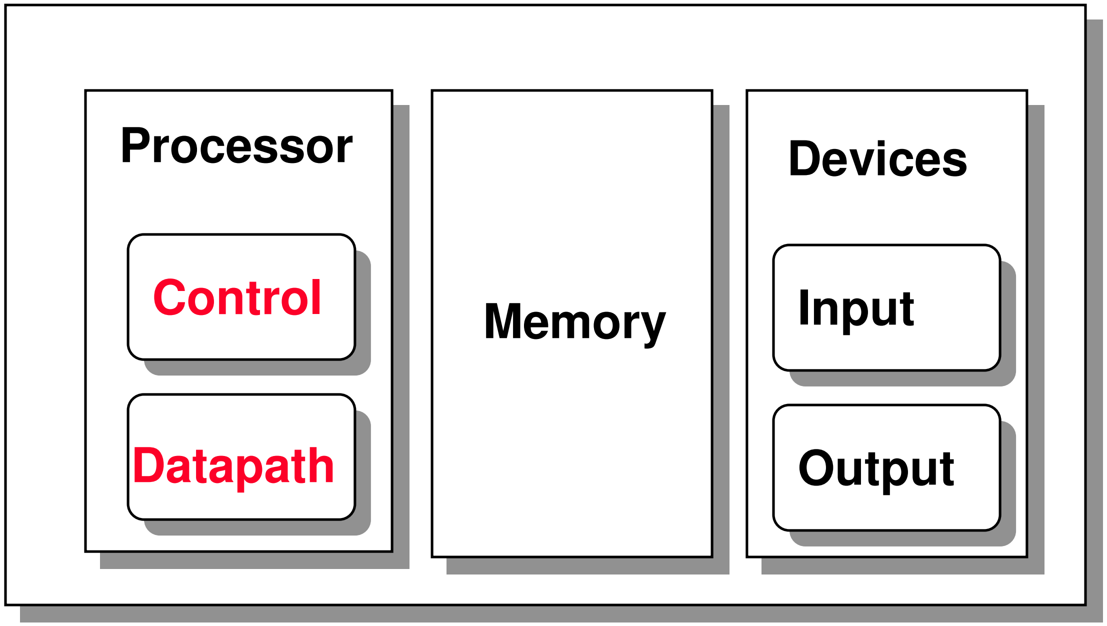
- Registers are
  - Faster than main memory
  - Easier for a compiler to use
    - e.g., (A*B) – (C*D) – (E*F) can do multiplies in any order vs. stack
  - Can hold variables so that
    - code density improves (since register are named with fewer bits than a memory location)
- Register addresses are indicated by using $

Naming Conventions for Registers

0	$zero	constant 0 (Hdware)
1	$at	reserved for assembler
2	$v0	expression evaluation &
3	$v1	function results
4	$a0	arguments
5	$a1	
6	$a2	
7	$a3	
8	$t0	temporary: caller saves
. . .		(callee can clobber)
15	$t7	
16	$s0	callee saves
. . .		(caller can clobber)
23	$s7	
24	$t8	temporary (cont’d)
25	$t9	
26	$k0	reserved for OS kernel
27	$k1	
28	$gp	pointer to global area
29	$sp	stack pointer
30	$fp	frame pointer
31	$ra	return address (Hdware)

Registers vs. Memory

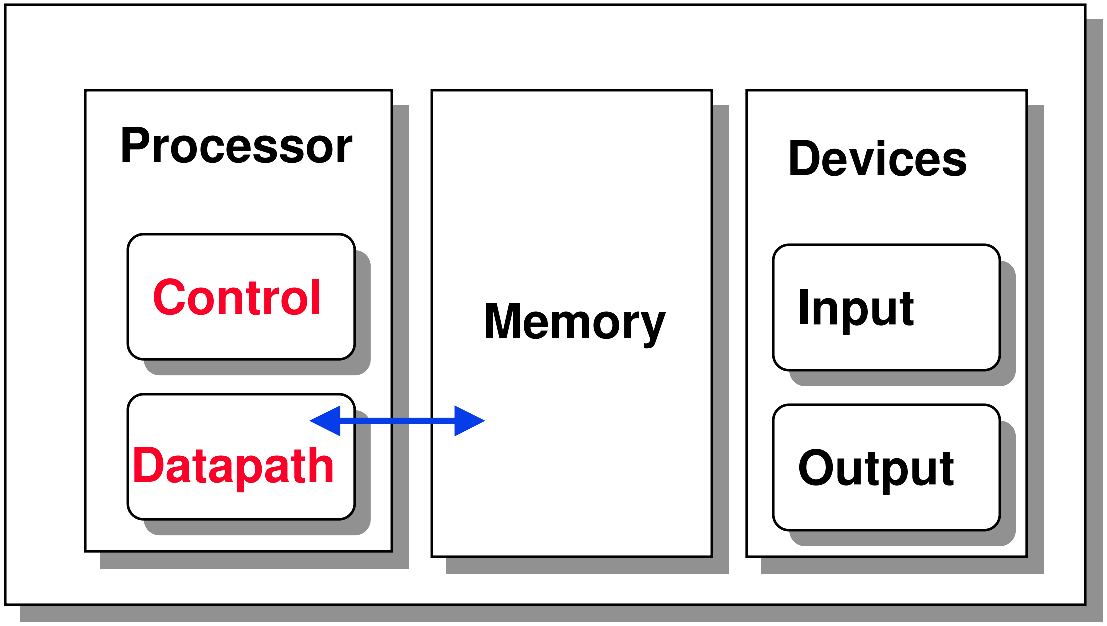
- Arithmetic instructions operands must be registers,
  — only thirty-two registers provided



- Compiler associates variables with registers
- What about programs with lots of variables?

Registers vs. Memory

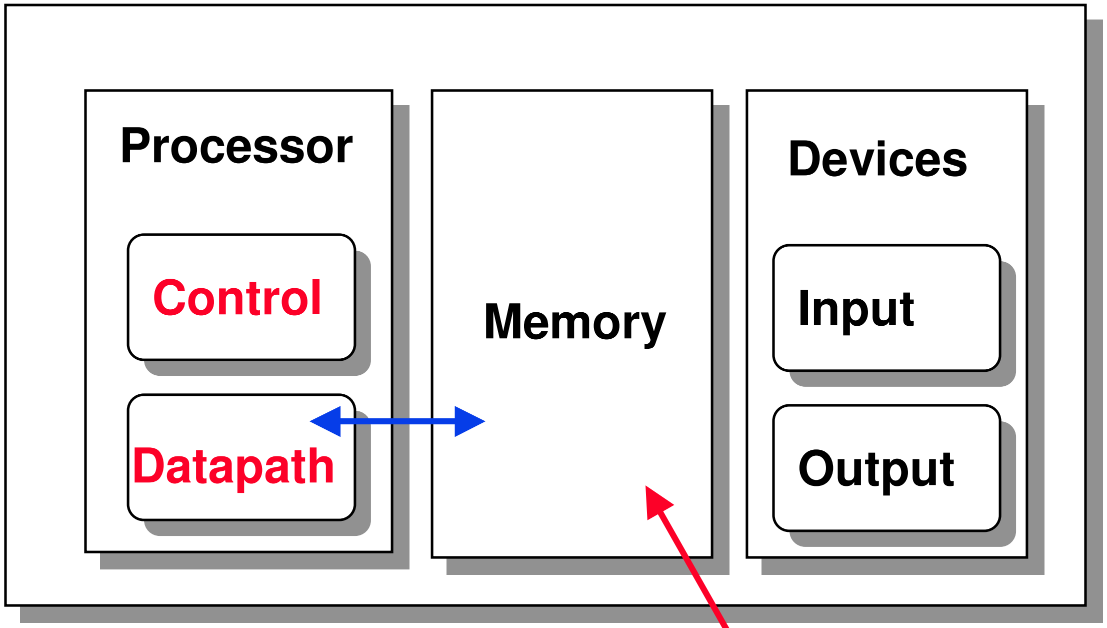
- Arithmetic instructions operands must be registers,
  — only thirty-two registers provided



- Compiler associates variables with registers
- What about programs with lots of variables?

Registers vs. Memory

- Arithmetic instructions operands must be registers,
  — only thirty-two registers provided



- Compiler associates variables with registers
- What about programs with lots of variables?

Accessing Memory

- MIPS has two basic data transfer instructions for accessing memory

```
lw    $t0, 4($s3)   #load word from memory

sw    $t0, 8($s3)   #store word to  memory
```

(assume `$s3` holds 24_{10})

- The data transfer instruction must specify
  - where in memory to read from (load) or write to (store) – memory address
  - where in the register file to write to (load) or read from (store) – register destination (source)
- The memory address is formed by summing the constant portion of the instruction and the contents of the second register

Accessing Memory

- ❑ MIPS has two basic data transfer instructions for accessing memory

```
                  28
lw    $t0, 4($s3)   #load word from memory

sw    $t0, 8($s3)   #store word to  memory
```

(assume `$s3` holds 24_{10})

- ❑ The data transfer instruction must specify
  - where in memory to read from (load) or write to (store) – memory address
  - where in the register file to write to (load) or read from (store) – register destination (source)
- ❑ The memory address is formed by summing the constant portion of the instruction and the contents of the second register

Accessing Memory

- MIPS has two basic data transfer instructions for accessing memory

```
                28
lw   $t0, 4($s3)   #load word from memory

sw   $t0, 8($s3)   #store word to  memory
                32
```

(assume `$s3` holds 24_{10})

- The data transfer instruction must specify
  - where in memory to read from (load) or write to (store) – memory address
  - where in the register file to write to (load) or read from (store) – register destination (source)
- The memory address is formed by summing the constant portion of the instruction and the contents of the second register

MIPS Instructions, so far

Category	Instr	Op Code	Example	Meaning
Arithmetic (R format)	add	0 and 32	add $s1, $s2, $s3	$s1 = $s2 + $s3
	subtract	0 and 34	sub $s1, $s2, $s3	$s1 = $s2 - $s3
Data transfer (I format)	load word	35	lw $s1, 100($s2)	$s1 = Memory($s2+100)
	store word	43	sw $s1, 100($s2)	Memory($s2+100) = $s1