Computer Architecture and Assembly Language

Prof. David August

COS 217

Goals of Today's Lecture

- Computer architecture
  - Central processing unit (CPU)
  - Fetch-decode-execute cycle
  - Memory hierarchy, and other optimization
- Assembly language
  - Machine vs. assembly vs. high-level languages
  - Motivation for learning assembly language
  - Intel Architecture (IA32) assembly language

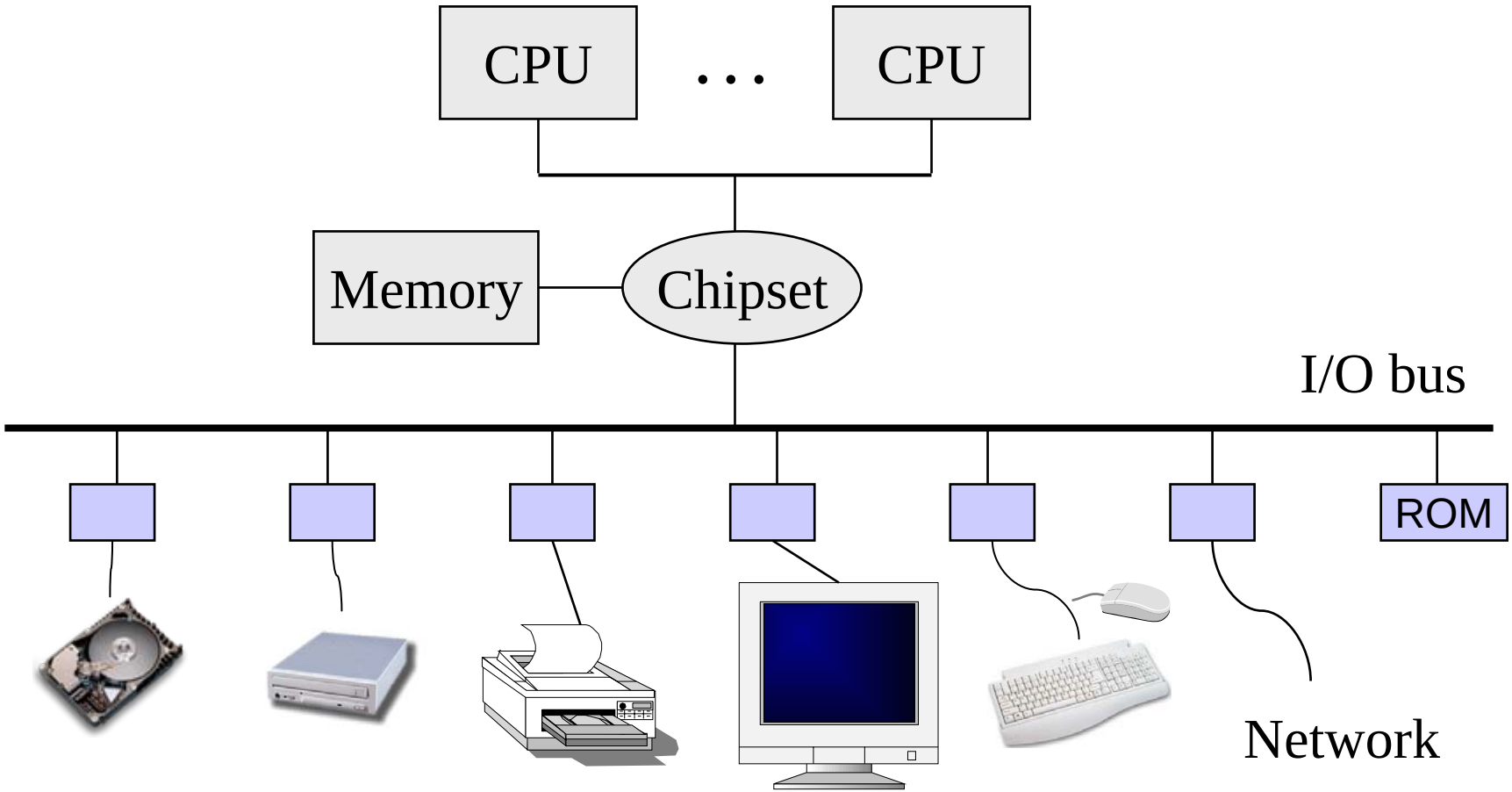
Levels of Languages

- Machine language
  - What the computer sees and deals with
  - Every command is a sequence of one or more numbers
- Assembly language
  - Command numbers replaced by letter sequences that are easier to read
  - Still have to work with the specifics of the machine itself
- High-level language
  - Make programming easier by describing operations in a natural language
  - A single command replaces a group of low-level assembly language commands

Why Learn Assembly Language?

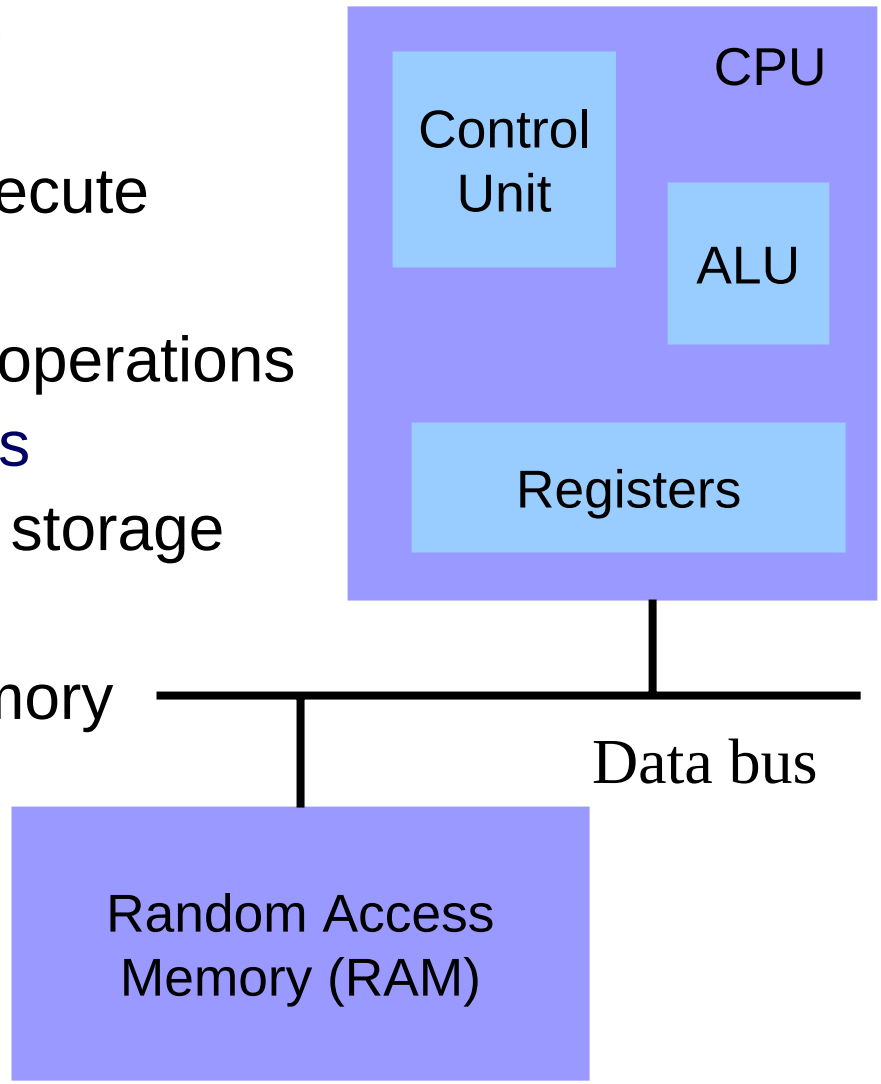
- Understand how things work underneath
  - Learn the basic organization of the underlying machine
  - Learn how the computer actually runs a program
  - Design better computers in the future
- Write faster code (even in high-level language)
  - By understanding which high-level constructs are better
  - … in terms of how efficient they are at the machine level
- Some software is still written in assembly language
  - Code that really needs to run quickly
  - Code for embedded systems, network processors, etc.

A Typical Computer



Von Neumann Architecture

- Central Processing Unit
  - Control unit
    - Fetch, decode, and execute
  - Arithmetic and logic unit
    - Execution of low-level operations
  - General-purpose registers
    - High-speed temporary storage
  - Data bus
    - Provide access to memory
- Memory
  - Store instructions
  - Store data



Control Unit

- Instruction pointer
  - Stores the location of the next instruction
    - Address to use when reading from memory
  - Changing the instruction pointer
    - Increment by one to go to the next instruction
    - Or, load a new value to “jump” to a new location
- Instruction decoder
  - Determines what operations need to take place
    - Translate the machine-language instruction
  - Control the registers, arithmetic logic unit, and memory
    - E.g., control which registers are fed to the ALU
    - E.g., enable the ALU to do multiplication
    - E.g., read from a particular address in memory

Example: Kinds of Instructions

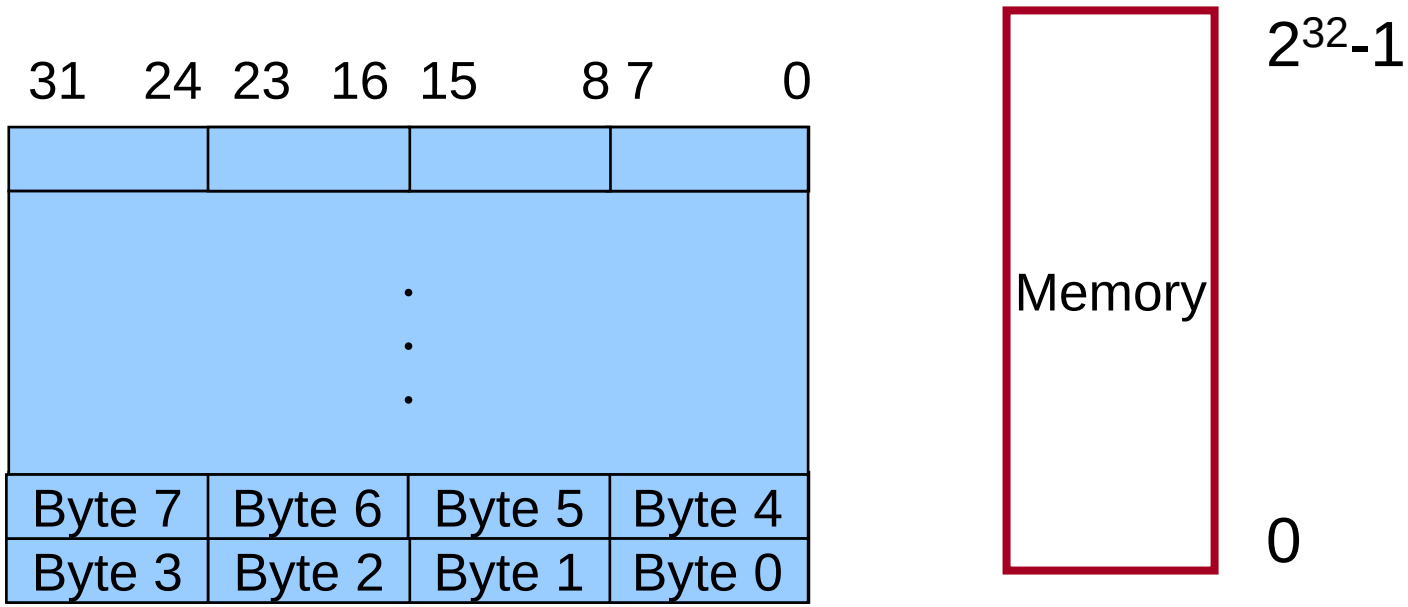
```
count = 0;
while (n > 1) {
  count++;
  if (n & 1)
    n = n*3 + 1;
  else
    n = n/2;
}
```

- **Storing values in registers**
  - count = 0
  - n
- **Arithmetic and logic operations**
  - Increment: count++
  - Multiply: n * 3
  - Divide: n/2
  - Logical AND: n & 1
- **Checking results of comparisons**
  - while (n > 1)
  - if (n & 1)
- **Jumping**
  - To the end of the while loop (if “n > 1”)
  - Back to the beginning of the loop
  - To the else clause (if “n & 1” is 0)

Size of Variables

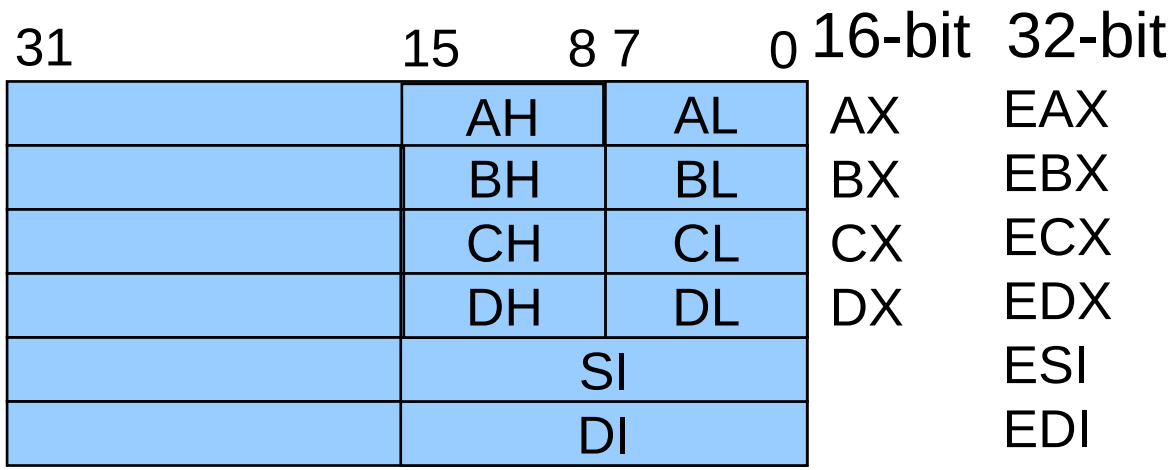
- Data types in high-level languages vary in size
  - Character: 1 byte
  - Short, int, and long: varies, depending on the computer
  - Pointers: typically 4 bytes
  - Struct: arbitrary size, depending on the elements
- Implications
  - Need to be able to store and manipulate in multiple sizes
  - Byte (1 byte), word (2 bytes), and extended (4 bytes)
  - Separate assembly-language instructions
    - e.g., addb, addw, addl
  - Separate ways to access (parts of) a 4-byte register

Four-Byte Memory Words



Byte order is little endian

IA32 General Purpose Registers



General-purpose registers

Registers for Executing the Code

- Execution control flow
  - Instruction pointer (EIP)
    - Address in memory of the current instruction
  - Flags (EFLAGS)
    - Stores the status of operations, such as comparisons
    - E.g., last result was positive/negative, was zero, etc.
- Function calls (more on these later!)
  - Stack register (ESP)
    - Address of the top of the stack
  - Base pointer (EBP)
    - Address of a particular element on the stack
    - Access function parameters and local variables

Other Registers that you don't much care about

- Segment registers
  - CS, SS, DS, ES, FS, GS
- Floating Point Unit (FPU) (x87)
  - Eight 80-bit registers (ST0, …, ST7)
  - 16-bit control, status, tag registers
  - 11-bit opcode register
  - 48-bit FPU instruction pointer, data pointer registers
- MMX
  - Eight 64-bit registers
- SSE and SSE2
  - Eight 128-bit registers
  - 32-bit MXCRS register
- System
  - I/O ports
  - Control registers (CR0, …, CR4)
  - Memory management registers (GDTR, IDTR, LDTR)
  - Debug registers (DR0, …, DR7)
  - Machine specific registers
  - Machine check registers
  - Performance monitor registers

Reading IA32 Assembly Language

- Assembler directives: starting with a period (“.”)
  - E.g., “.section .text” to start the text section of memory
  - E.g., “.loop” for the address of an instruction
- Referring to a register: percent size (“%”)
  - E.g., “%ecx” or “%eip”
- Referring to a constant: dollar sign (“$”)
  - E.g., “$1” for the number 1
- Storing result: typically in the second argument
  - E.g. “addl $1, %ecx” increments register ECX
  - E.g., “movl %edx, %eax” moves EDX to EAX
- Comment: pound sign (“#”)
  - E.g., “# Purpose: Convert lower to upper case”

Detailed Example

n %edx
count %ecx

```
count=0;
while (n>1) {
   count++;
   if (n&1)
      n = n*3+1;
   else
      n = n/2;
}
```

```
        movl   $0, %ecx
.loop:
        cmpl   $1, %edx
        jle    .endloop
        addl   $1, %ecx
        movl   %edx, %eax
        andl   $1, %eax
        je     .else
        movl   %edx, %eax
        addl   %eax, %edx
        addl   %eax, %edx
        addl   $1, %edx
        jmp    .endif
.else:
        sarl   $1, %edx
.endif:
        jmp    .loop
.endloop:
```

Flattening Code Example

```
count=0;
while (n>1) {
  count++;
  if (n&1)
    n = n*3+1;
  else
    n = n/2;
}
```

Machine-Language Instructions

Instructions have the form

op source, dest "dest ← dest ⊕ source"

operation (move, add, subtract, etc.)

first operand (and destination)

second operand

Instruction Format:

opcode	operand	operand

Instruction

- Opcode
  - What to do
- Source operands
  - Immediate (in the instruction itself)
  - Register
  - Memory location
  - I/O port
- Destination operand
  - Register
  - Memory location
  - I/O port
- Assembly syntax

  Opcode source1, [source2,] destination

How Many Instructions to Have?

- Need a certain minimum set of functionality
  - Want to be able to represent any computation that can be expressed in a higher-level language
- Benefits of having many instructions
  - Direct implementation of many key operations
  - Represent a line of C in one (or just a few) lines of assembly
- Disadvantages of having many instructions
  - Larger opcode size
  - More complex logic to implement complex instructions
  - Hard to write compilers to exploit all the available instructions
  - Hard to optimize the implementation of the CPU

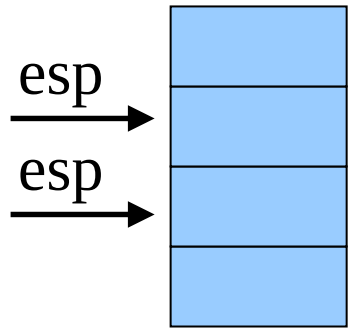
CISC vs. RISC

Complex Instruction Set Computer	Reduced Instruction Set Computer
(old fashioned, 1970s style)	(“modern”, 1980s style)
Examples:	Examples:
Vax (1978-90)	MIPS (1985-?)
Motorola 68000 (1979-90)	Sparc (1986-2006)
8086/80x86/Pentium (1974-2025)	IBM PowerPC (1990-?)
	ARM
Instructions of various lengths, designed to economize on memory (size of instructions)	Instructions all the same size and all the same format, designed to economize on decoding complexity (and time, and power drain)

Data Transfer Instructions

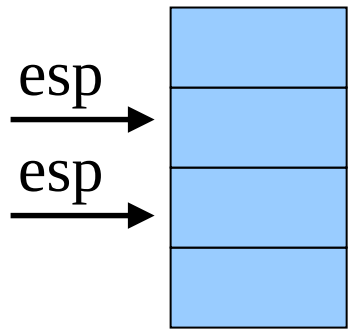
- **`mov{b,w,l} source, dest`**
  - General move instruction
- **`push{w,l} source`**

```
pushl %ebx    # equivalent instructions
              subl $4, %esp
              movl %ebx, (%esp)
```



- **`pop{w,l} dest`**

```
popl %ebx     # equivalent instructions
              movl (%esp), %ebx
              addl  $4, %esp
```

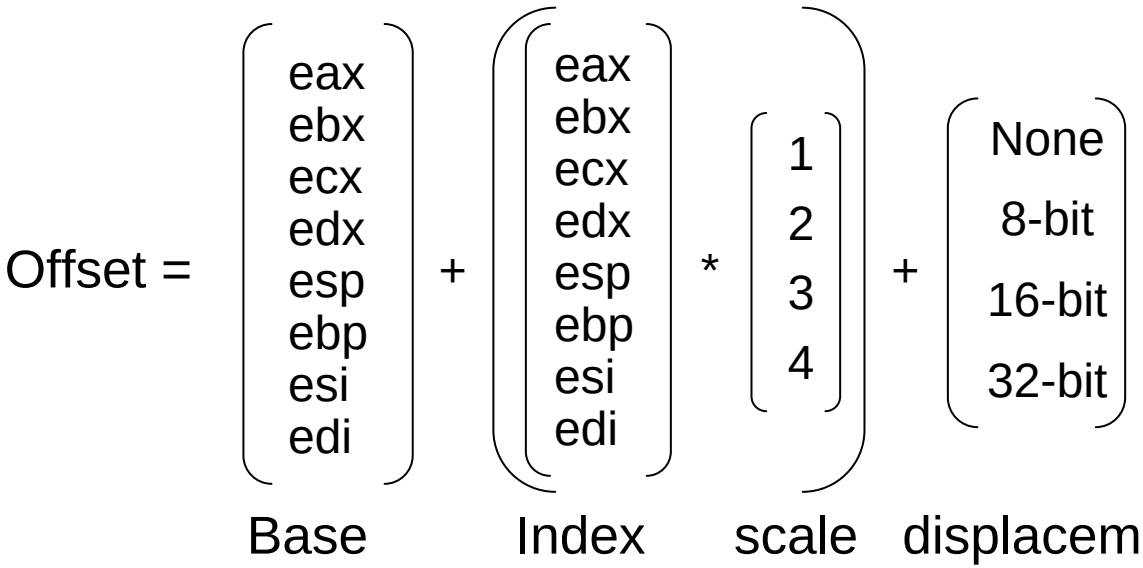


- Many more in Intel manual (volume 2)
  - Type conversion, conditional move, exchange, compare and exchange, I/O port, string move, etc.

Data Access Methods

- Immediate addressing: data stored in the instruction itself
  - **movl $10, %ecx**
- Register addressing: data stored in a register
  - **movl %eax, %ecx**
- Direct addressing: address stored in instruction
  - **movl 2000, %ecx**
- Indirect addressing: address stored in a register
  - **movl (%eax), %ebx**
- Base pointer addressing: includes an offset as well
  - **movl 4(%eax), %ebx**
- Indexed addressing: instruction contains base address, and specifies an index register and a multiplier (1, 2, or 4)
  - **movl 2000(,%ecx,1), %ebx**

Effective Address



- Displacement `movl foo, %ebx`
- Base `movl (%eax), %ebx`
- Base + displacement `movl foo(%eax), %ebx` `movl 1(%eax), %ebx`
- (Index * scale) + displacement `movl (,%eax,4), %ebx`
- Base + (index * scale) + displacement `movl foo(,%eax,4), %ebx`

Bitwise Logic Instructions

- Simple instructions

and{b,w,l} source, dest	dest = source & dest
or{b,w,l} source, dest	dest = source | dest
xor{b,w,l} source, dest	dest = source ^ dest
not{b,w,l} dest	dest = ^dest
sal{b,w,l} source, dest (arithmetic)	dest = dest << source
sar{b,w,l} source, dest (arithmetic)	dest = dest >> source

- Many more in Intel Manual (volume 2)
  - Logic shift
  - Rotation shift
  - Bit scan
  - Bit test
  - Byte set on conditions

Arithmetic Instructions

- Simple instructions
  - add{b,w,l} source, dest — dest = source + dest
  - sub{b,w,l} source, dest — dest = dest – source
  - inc(b,w,l} dest — dest = dest + 1
  - dec{b,w,l} dest — dest = dest – 1
  - neg(b,w,l} dest — dest = ^dest
  - cmp{b,w,l} source1, source2 — source2 – source1
- Multiply
  - mul (unsigned) or imul (signed)

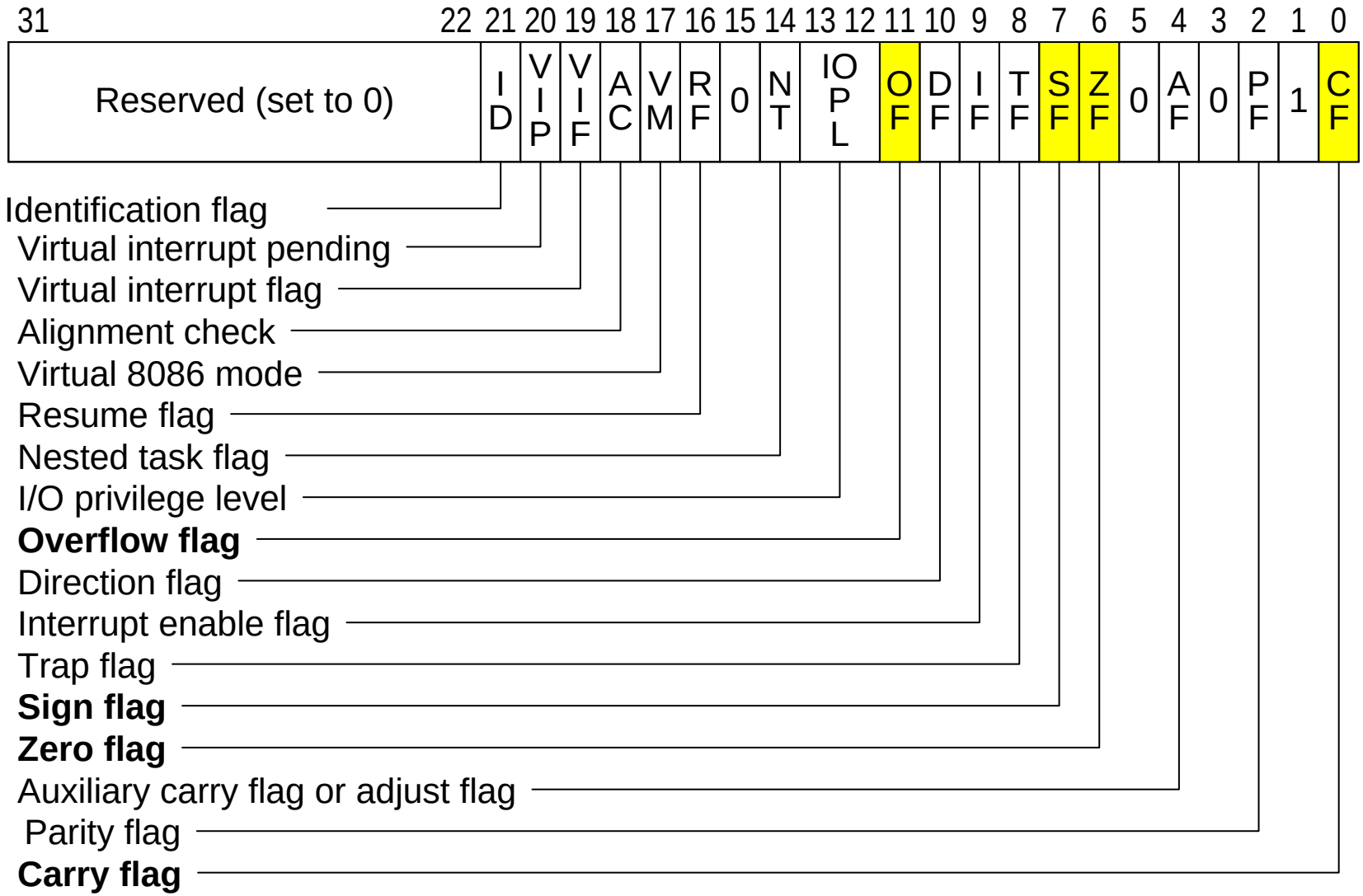
```
mull %ebx          # edx, eax = eax * ebx
```

- Divide
  - div (unsigned) or idiv (signed)

```
idiv %ebx          # edx = edx,eax / ebx
```

- Many more in Intel manual (volume 2)
  - adc, sbb, decimal arithmetic instructions

EFLAG Register & Condition Codes



Branch Instructions

- Conditional jump
  - j{l,g,e,ne,...} target if (condition) {eip = target}

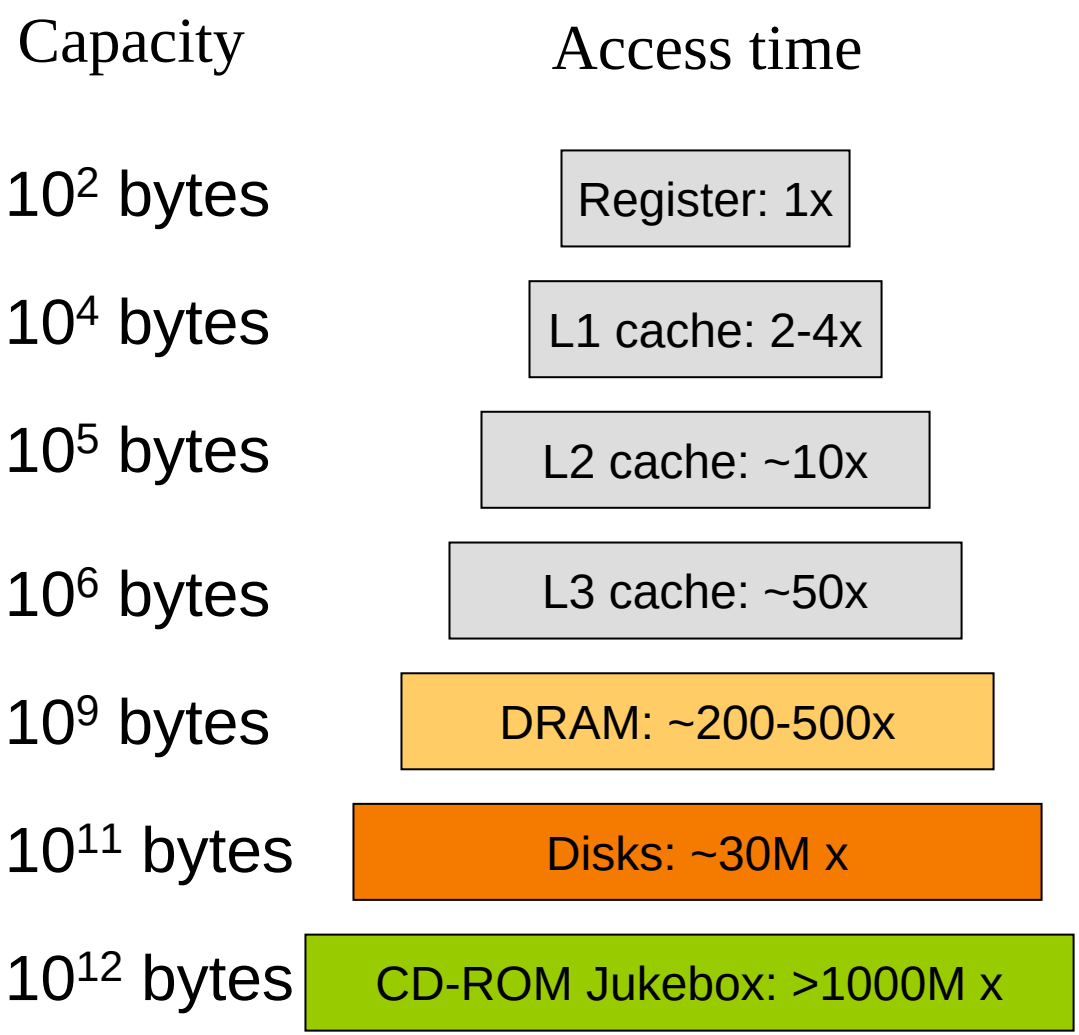
Comparison	Signed	Unsigned	
=	e	e	*"equal"*
≠	ne	ne	*"not equal"*
>	g	a	*"greater,above"*
≥	ge	ae	*"...-or-equal"*
<	l	b	*"less,below"*
≤	le	be	*"...-or-equal"*
overflow/carry	o	c	
no ovf/carry	no	nc	

- Unconditional jump
  - jmp target
  - jmp *register

Making the Computer Faster

- Memory hierarchy
  - Ranging from small, fast storage to large, slow storage
  - E.g., registers, caches, main memory, disk, CDROM, …
- Sophisticated logic units
  - Have dedicated logic units for specialized functions
  - E.g., right/left shifting, floating-point operations, graphics, network,…
- Pipelining
  - Overlap the fetch-decode-execute process
  - E.g., execute instruction i, while decoding i-1, and fetching i-2
- Branch prediction
  - Guess which way a branch will go to avoid stalling the pipeline
  - E.g., assume the “for loop” condition will be true, and keep going
- And so on… see the Computer Architecture class!

Memory Hierarchy



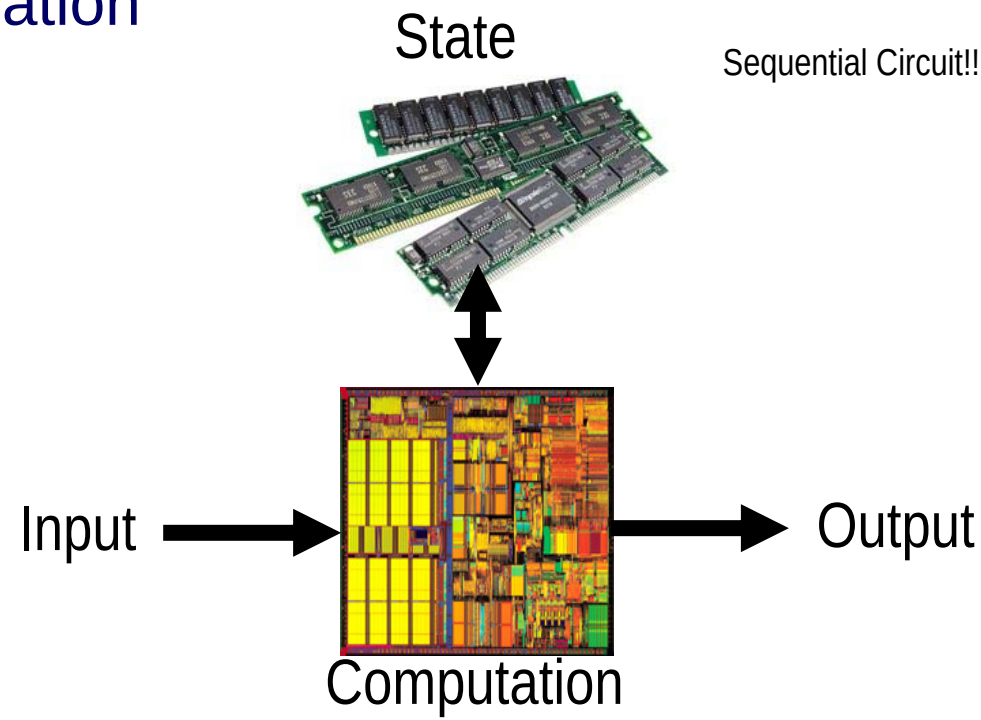
Conclusion

- Computer architecture
  - Central Processing Unit (CPU) and Random Access Memory (RAM)
  - Fetch-decode-execute cycle
  - Instruction set
- Assembly language
  - Machine language represented with handy mnemonics
  - Example of the IA-32 assembly language
- Next time
  - Portions of memory: data, bss, text, stack, etc.
  - Function calls, and manipulating contents of the stack

Instructions

Computers process information

- Input/Output (I/O)
- State (memory)
- Computation (processor)



- Instructions instruct processor to manipulate state
- Instructions instruct processor to produce I/O in the same way

State

Typical modern machine has this architectural state:

1. Main Memory
2. Registers
3. Program Counter

Architectural – Part of the assembly programmer's interface

(Implementation has additional microarchitectural state)

State – Main Memory

Main Memory (AKA: RAM – Random Access Memory)

- Data can be accessed by address (like a big array)
- Large but relatively slow
- Decent desktop machine: 1 Gigabyte, 800MHz

Address	Data
0000	01011001_2
0001	$F5_{16}$
0002	78_{16}
0003	$3A_{16}$
…	…
FFFF	00000000_2

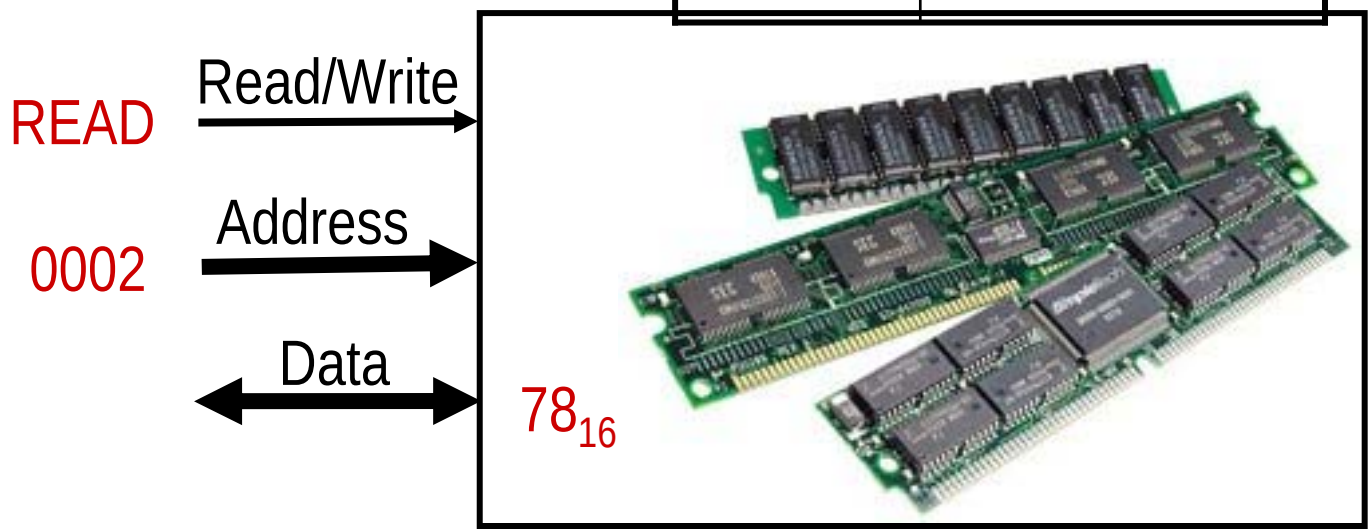
Byte Addressable

State – Main Memory

Read:

1. Indicate READ
2. Give Address
3. Get Data

Address	Data
0000	01011001_2
0001	$F5_{16}$
0002	78_{16}
0003	$3A_{16}$
…	…
FFFF	00000000_2

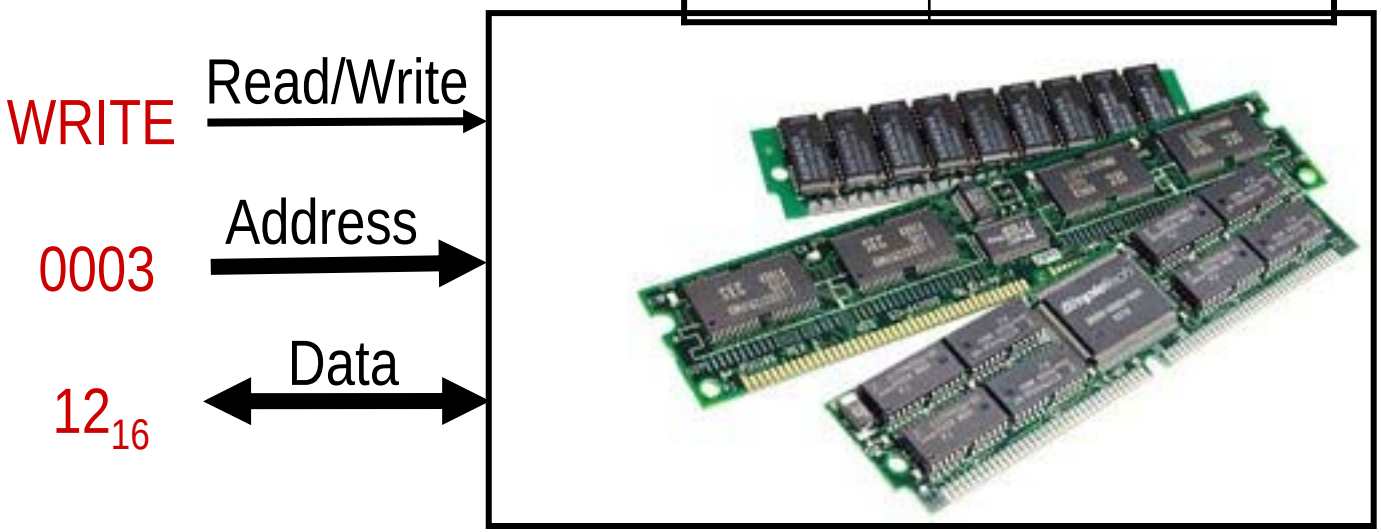


State – Main Memory

Write:

1. Indicate WRITE
2. Give Address and Data

Address	Data
0000	01011001_2
0001	$F5_{16}$
0002	12_{16}
0003	$3A_{16}$
…	…
FFFF	00000000_2

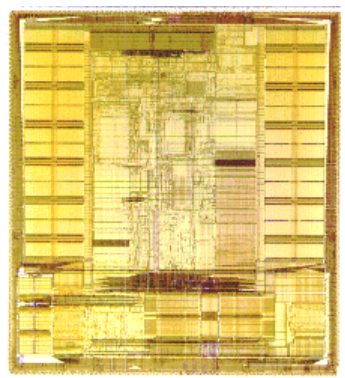


State – Registers (Register File)

Data can be accessed by register number (address)

- Small but relatively fast (typically on processor chip)
- Decent desktop machine: 8 32-bit registers, 3 GHz

Register	Data in Reg
0	00000000_{16}
1	$F629D9B5_{16}$
2	$7B2D9D08_{16}$
3	00000001_{16}
…	…
8	$DEADBEEF_{16}$



State – Program Counter

Program Counter (AKA: PC, Instruction Pointer, IP)

- Instructions change state, but which instruction now?
- PC holds memory address of currently executing instruction

Program Counter
0002

Address	Data in Memory
0000	01011001_2
0001	$F5_{16}$
0002	ADD_{inst}
0003	$SUBTRACT_{inst}$
…	…
FFFF	00000000_2

State – Program Counter

Program Counter (AKA: PC, Instruction Pointer, IP)

- Instructions change state, but which instruction now?
- PC holds address of currently executing instruction
- PC is updated after each instruction

Program Counter
0003

Address	Data in Memory
0000	01011001_2
0001	$F5_{16}$
0002	ADD_{inst}
0003	$SUBTRACT_{inst}$
…	…
FFFF	00000000_2

State – Summary

Typical modern machine has this architectural state:

1. Main Memory – Big, Slow
2. Registers – Small, Fast (always on processor chip)
3. Program Counter – Address of executing instruction

Architectural – Part of the assembly programmer's interface

(implementation has additional microarchitectural state)

An Aside: State and The Core Dump

- Core Dump: the state of the machine at a given time
- Typically at program failure
- Core dump contains:
  - Register Contents
  - Memory Contents
  - PC Value

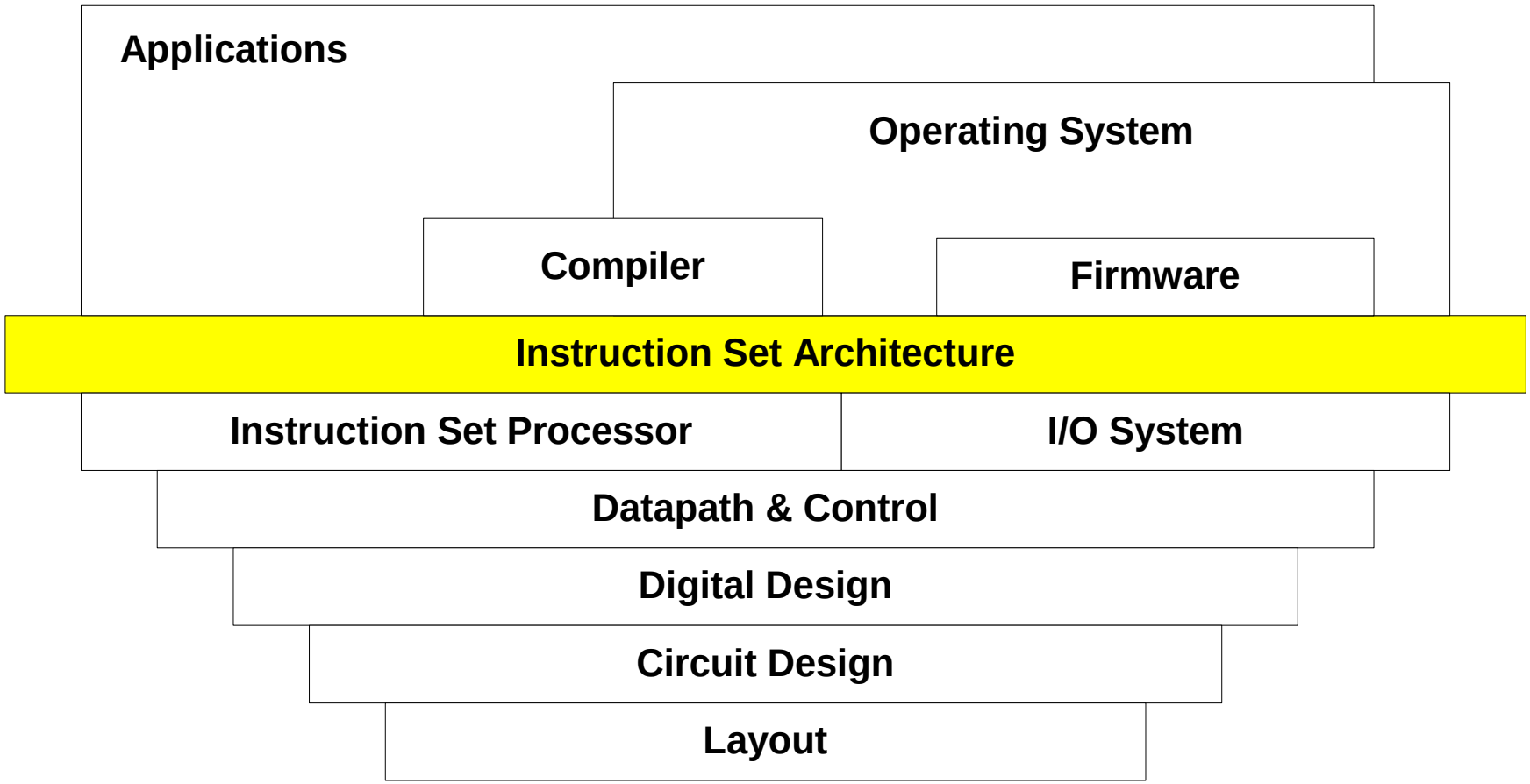
PC
10

Registers							
0	1	2	3	4	5	6	7
0000	0788	B700	0010	0401	0002	0003	00A0
8	9	A	B	C	D	E	F
0000	0788	B700	0010	0401	0002	0003	00A0

Main Memory	
00:	0000 0000 0000 0000 0000 0000 0000 0000
08:	0000 0000 0000 0000 0000 0000 0000 0000
10:	9222 9120 1121 A120 1121 A121 7211 0000
18:	0000 0001 0002 0003 0004 0005 0006 0007
20:	0008 0009 000A 000B 000C 000D 000E 000F
28:	0000 0000 0000 FE10 FACE CAFE ACED CEDE
.	
.	
E8:	1234 5678 9ABC DEF0 0000 0000 F00D 0000
F0:	0000 0000 EEEE 1111 EEEE 1111 0000 0000
F8:	B1B2 F1F5 0000 0000 0000 0000 0000 0000

Interfaces in Computer Systems

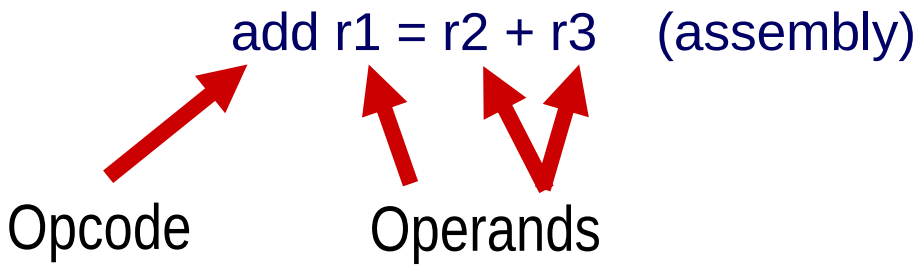
Software: Produce Bits Instructing Machine to Manipulate State or Produce I/O



Hardware: Read and Obey Instruction Bits

Instructions

An ADD Instruction:



Parts of the Instruction:

- Opcode (verb) – what operation to perform
- Operands (noun) – what to operate upon
- Source Operands – where values come from
- Destination Operand – where to deposit data values

Instructions

Instructions:

"The vocabulary of commands"

Specify how to operate on state

Register	Data
0	0
1	15
2	1
3	2
…	…
31	0

Example:

40: add r1 = r2 + r3
44: sub r3 = r1 - r0
48: store M[r3] = r1
52: load r2 = M[2]

Program Counter
40

Address	Data
0	0
1	25
2	5
3	9
…	…
FFFFFFFF	0

Instructions

Instructions:

"The vocabulary of commands"

Specify how to operate on state

Register	Data
0	0
1	15
2	**1**
3	**2**
…	…
31	0

Example: **3**

40: add r1 = r2 + r3
44: sub r3 = r1 - r0
48: store M[r3] = r1
52: load r2 = M[2]

Program Counter
40

Address	Data
0	0
1	25
2	5
3	9
…	…
FFFFFFFF	0

Instructions

Register	Data
0	0
1	3
2	1
3	2
...	...
31	0

Example:

40: add r1 = r2 + r3

44: sub r3 = r1 - r0

48: store M[r3] = r1

52: load r2 = M[2]

Program Counter
40

Address	Data
0	0
1	25
2	5
3	9
...	...
FFFFFFFF	0

Instructions

Instructions:

"The vocabulary of commands"

Specify how to operate on state

Register	Data
0	0
1	3
2	1
3	2
…	…
31	0

Example:

40: add r **3** r2 + r3

44: sub r3 = r1 - r0

48: store M[r3] = r1

52: load r2 = M[2]

Program Counter
44

Address	Data
0	0
1	25
2	5
3	9
…	…
FFFFFFFF	0

Instructions

Instructions:

"The vocabulary of commands"

Specify how to operate on state

Register	Data
0	0
1	**3**
2	1
3	**3**
…	…
31	0

Example:

40: add r1 = r2 + r3

44: sub r3 = r1 - r0 **3**

48: store M[r3] = r1

52: load r2 = M[2]

Program Counter
48

Address	Data
0	0
1	25
2	5
3	9
…	…
FFFFFFFF	0

Instructions

Instructions:

"The vocabulary of commands"

Specify how to operate on state

Register	Data
0	0
1	3
2	1
3	3
…	…
31	0

Example:

40: add r1 = r2 + r3
44: sub r3 = r1 - r0
48: store **5** r3] = r1
52: load r2 = M[2]

Program Counter
52

Address	Data
0	0
1	25
2	**5**
3	3
…	…
FFFFFFFF	0

Instructions

Instructions:

"The vocabulary of commands"

Specify how to operate on state

Register	Data
0	0
1	3
2	5
3	3
…	…
31	0

Example:

40: add r1 = r2 + r3
44: sub r3 = r1 - r0
48: store M[r3] = r1
52: load r2 = M[2]

Program Counter
52

Address	Data
0	0
1	25
2	5
3	3
…	…
FFFFFFFF	0

Instructions

Note:

1. Insts Executed in Order
2. Addressing Modes

Example:

40: add r1 = r2 + r3
44: sub r3 = r1 - r0
48: store M[r3] = r1
52: load r2 = M[2]

Program Counter
52

Register	Data
0	0
1	3
2	5
3	3
…	…
31	0

Address	Data
0	0
1	25
2	5
3	3
…	…
FFFFFFFF	0

Assembly Instructions and C

```
add r1 = r2 + r3

sub r3 = r1 - r0

store M[ r3 ] = r1

load r2 = M[ 2 ]
```

```
main() {
  int a = 15, b = 1, c = 2;

  a = b + c;  /* a gets 3 */

  c = a;  /* c gets 3 */

  *(int *)c = a;
            /* M[c] = a */

  b = *(int *)(2);
            /* b gets M[2] */
}
```

Branching

Suppose we could only execute instructions in sequence.

Recall from our example:

40: add r1 = r2 + r3
44: sub r3 = r1 - r0
48: store M[r3] = r1
52: load r2 = M[2]

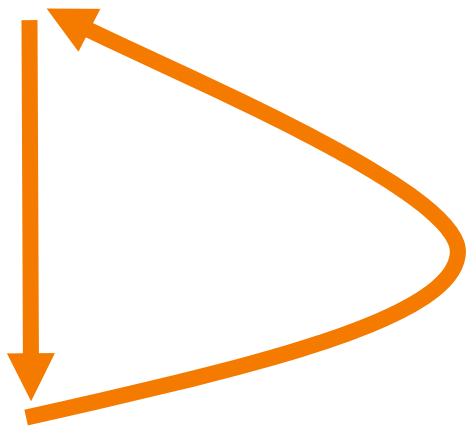
- In a decent desktop machine, how long would the longest program stored in main memory take?
- Assume: 1 instruction per cycle
  - An instruction is encoded in 4 bytes (32 bits)

Therefore…

- Some instructions must execute more than once
- PC must be updated

Example:

40: add r1 = r2 + r3
44: sub r3 = r1 - r0
48: store M[r3] = r1
52: load r2 = M[2]
56: PC = 40

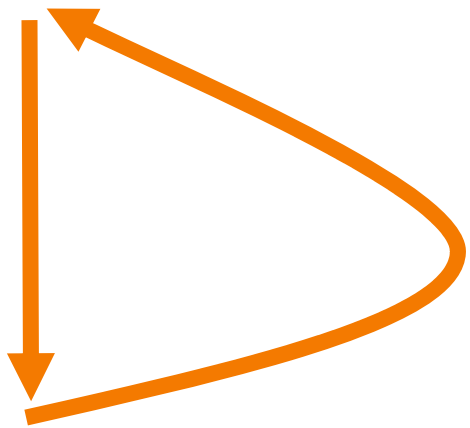


Unconditional Branches

- Unconditional branches always update the PC
- AKA: Jump instructions

Example:

```
40: add r1 = r2 + r3
44: sub r3 = r1 - r0
48: store M[ r3 ] = r1
52: load r2 = M[ 2 ]
56: jump 40
```



- How long with the program take?

Conditional Branch

- Conditional Branch sometimes updates PC
- AKA: Branch, Conditional Jump

- Example

```
40: r1 = 10
44: r1 = r1 - 1
48: branch r1 > 0, 44        if r1 is greater than 0, PC = 44
52: halt
```

- How long will this program take?

Conditional Branch

- What does this look like in C?

- Example

```
10: “Hello\n”          ; data in memory
36: arg1 = 10          ; argument memory address is 10
40: r1 = 10
44: r1 = r1 - 1
48: call printf         ; printf(arg1)
52: branch r1 > 0, 44
56: halt
```

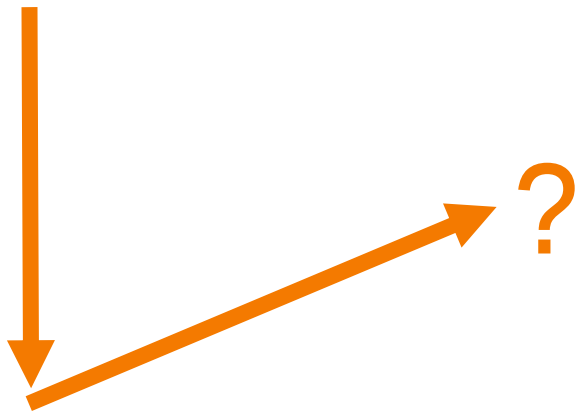
Details about red instructions/data next time…

Indirect Branches

- Branch address may also come from a register
- AKA: Indirect Jump

Example:

```
40: add r1 = r2 + r3
44: sub r3 = r1 - r0
48: store M[ r3 ] = r1
52: load r2 = M[ 2 ]
56: jump r4
60: halt
```



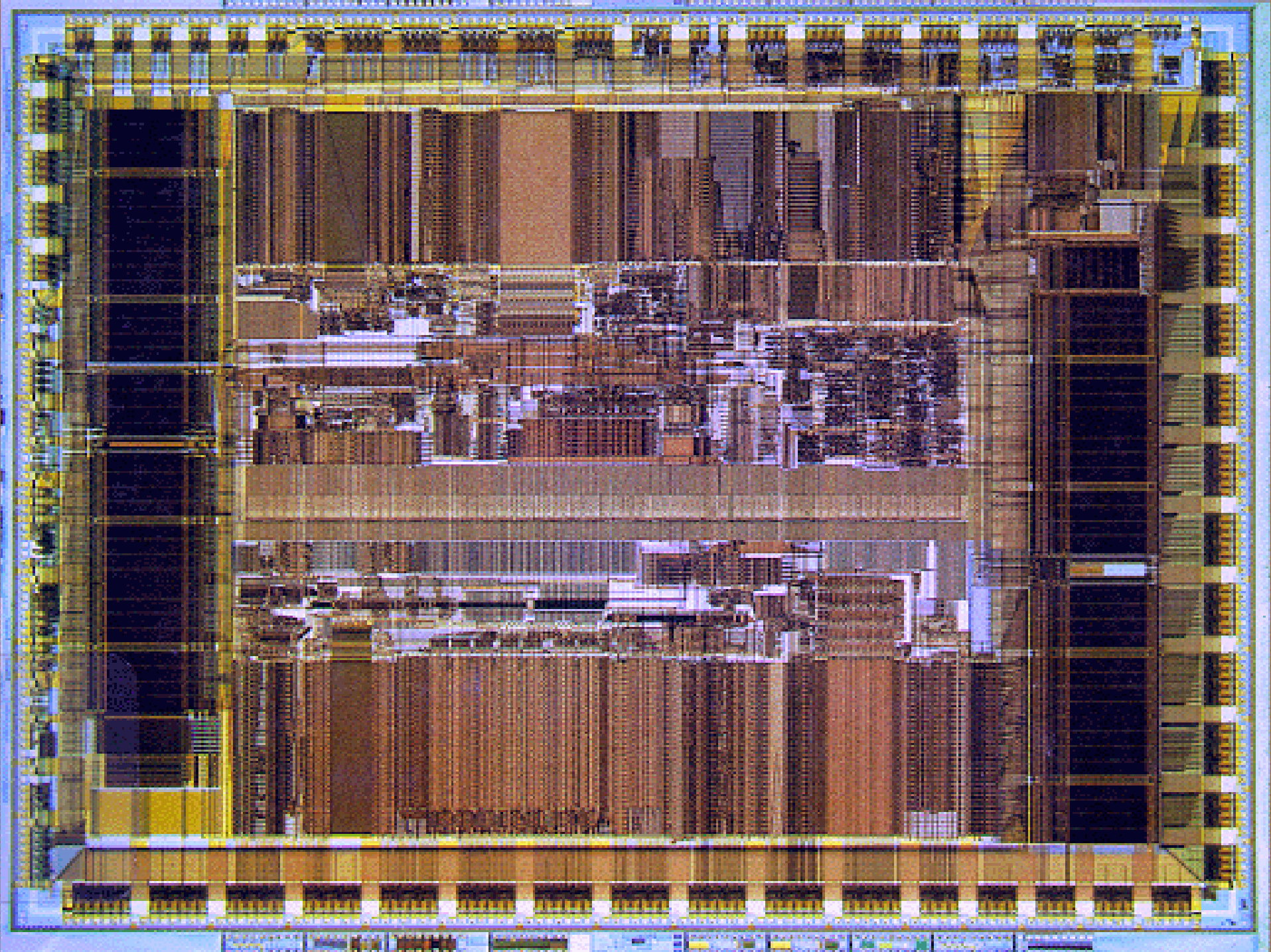
Branch Summary

- Reduce, Reuse, Recycle (instructions)
- Branch instructions update state

PC
10

Registers							
0	1	2	3	4	5	6	7
0000	0788	B700	0010	0401	0002	0003	00A0
8	9	A	B	C	D	E	F
0000	0788	B700	0010	0401	0002	0003	00A0

Main Memory	
00:	0000 0000 0000 0000 0000 0000 0000 0000
08:	0000 0000 0000 0000 0000 0000 0000 0000
10:	9222 9120 1121 A120 1121 A121 7211 0000
18:	0000 0001 0002 0003 0004 0005 0006 0007
20:	0008 0009 000A 000B 000C 000D 000E 000F
28:	0000 0000 0000 FE10 FACE CAFE ACED CEDE
.	
.	
E8:	1234 5678 9ABC DEF0 0000 0000 F00D 0000
F0:	0000 0000 EEEE 1111 EEEE 1111 0000 0000
F8:	B1B2 F1F5 0000 0000 0000 0000 0000 0000



A Note on Notation…

- Assembly syntax is somewhat arbitrary
- Equivalent “Add” Instructions
  - add r1, r2, r3
  - add r1 = r2, r3
  - r1 = r2 + r3
  - add r1 = r2 + r3
  - add $1, $2, $3
  - …
- Equivalent “Store Word” Instructions
  - sw $1, 10($2)
  - M[r2 + 10] = r1
  - st.w M[r2 + 10] = r1
  - …

Specific Instance: MIPS Instruction Set

- MIPS – SGI Workstations, Nintendo, Sony…

State:

- 32-bit addresses to memory (32-bit PC)
- 32 32-bit Registers
- A “word” is 32-bits on MIPS
- Register $0 ($zero) always has the value 0
- By convention, certain registers are used for certain things – more next time…

Specific Instance: MIPS Instruction Set

Some Arithmetic Instructions:

- Add:
  - Assembly Format: add <dest>, <src1>, <src2>
  - Example: add $1, $2, $3
  - Example Meaning: r1 = r2 + r3

- Subtract:
  - Same as add, except “sub” instead of “add”

Specific Instance: MIPS Instruction Set

Some Memory Instructions:

- Load Word:
  - Assembly Format: lw <dest>, <offset immediate> (<src1>)
  - Example: lw $1, 100 ($2)
  - Example Meaning: r1 = M[r2 + 100]

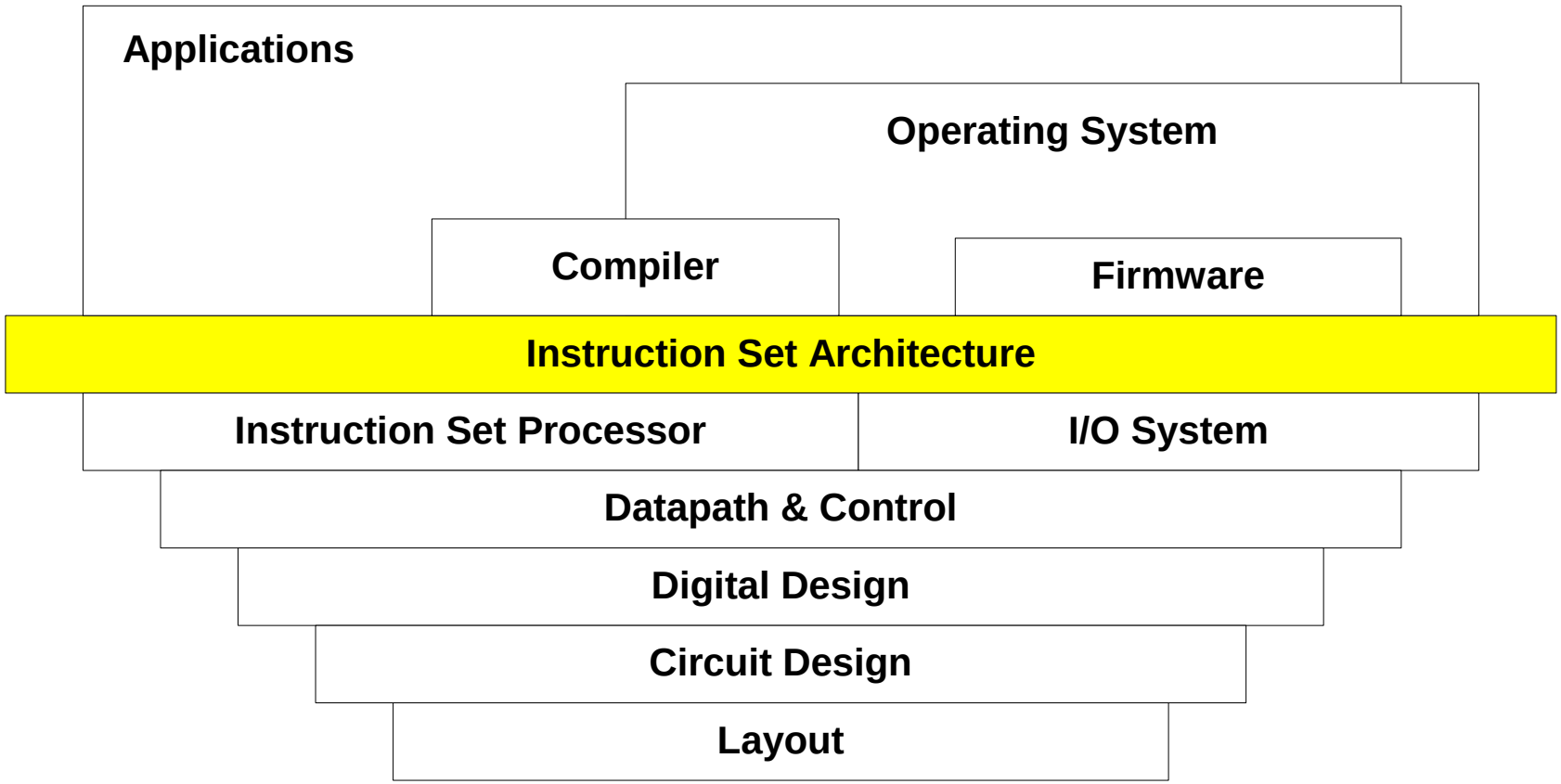
- Store Word:
  - Assembly Format: sw <src1>, <offset immediate> (<src2>)
  - Example: sw $1, 100 ($2)
  - Example Meaning: M[r2 + 100] = r1

Specific Instance: MIPS Instruction Set

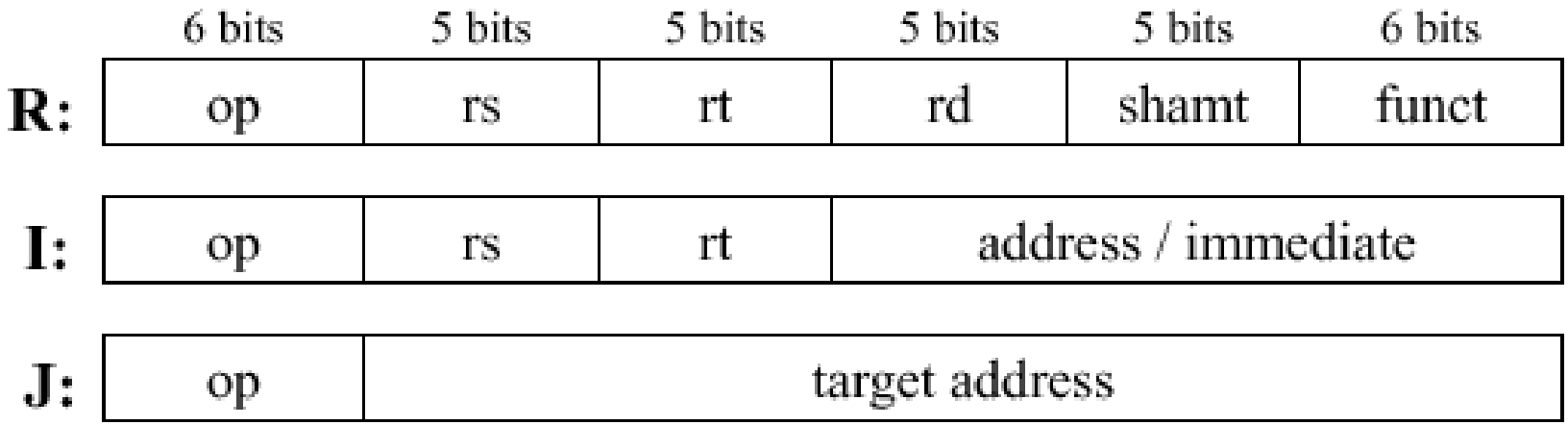
Some Branch Instructions:

- Branch Equal:
  - Assembly Format: beq <src1>, <src2>, <target immediate>
  - Example: beq $1, $2, 100
  - Example Meaning: branch r1 == r2, 100
    If r1 is equal to r2, PC = 100
- Branch Not Equal: Same except beq -> bne
- Jump:
  - Assembly Format: j <target immediate>
  - Example: j 100
  - Example Meaning: jump 100
    PC = 100

How are MIPS Instructions Encoded?

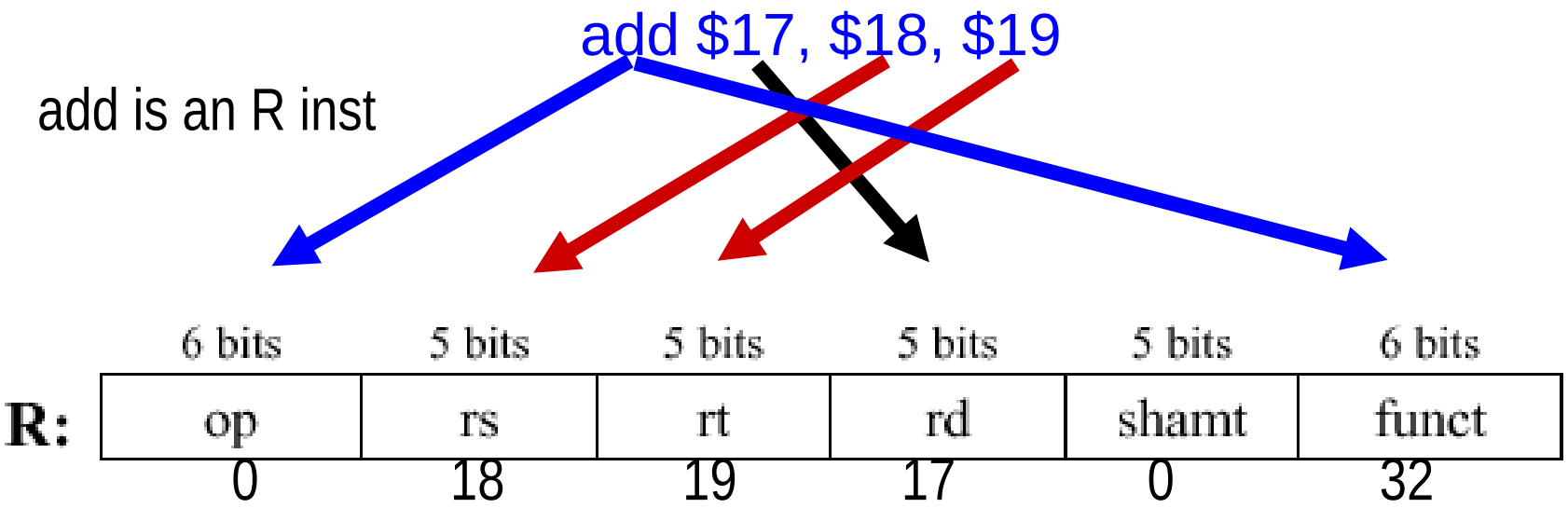


MIPS Encodings
32-bits/Instruction

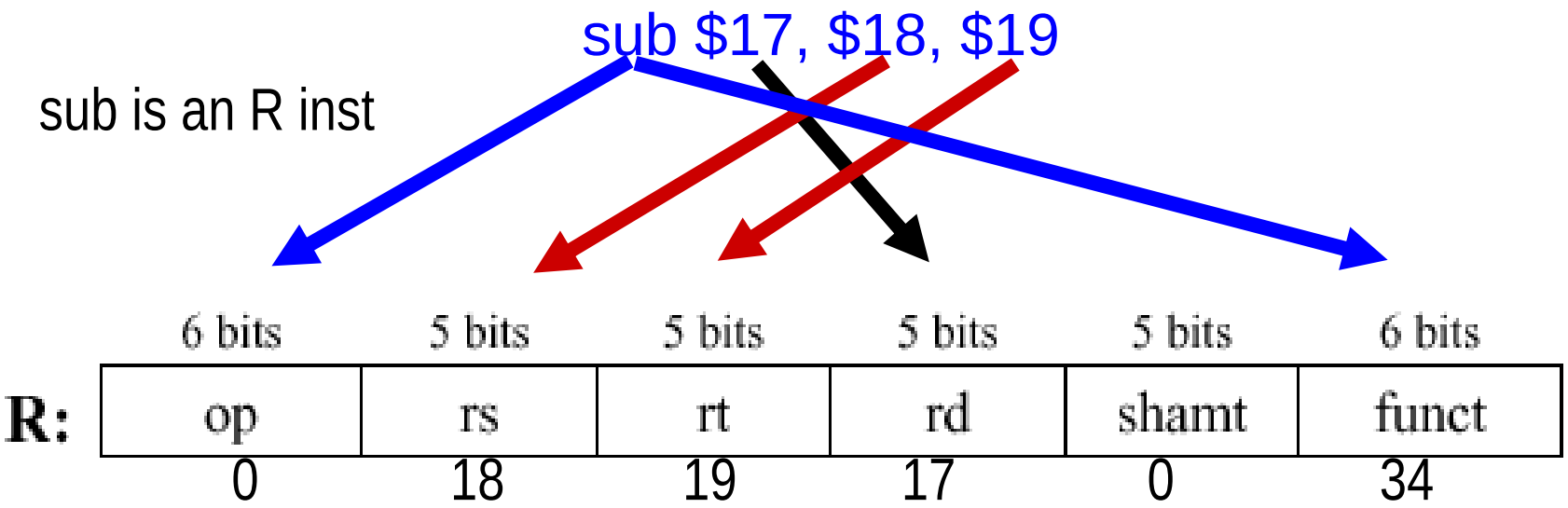


op: basic operation of the instruction (opcode)
rs: first source operand register
rt: second source operand register
rd: destination operand register
shamt: shift amount
funct: selects the specific variant of the opcode (function code)
address: offset for load/store instructions (+/-2^{15})
immediate: constants for immediate instructions

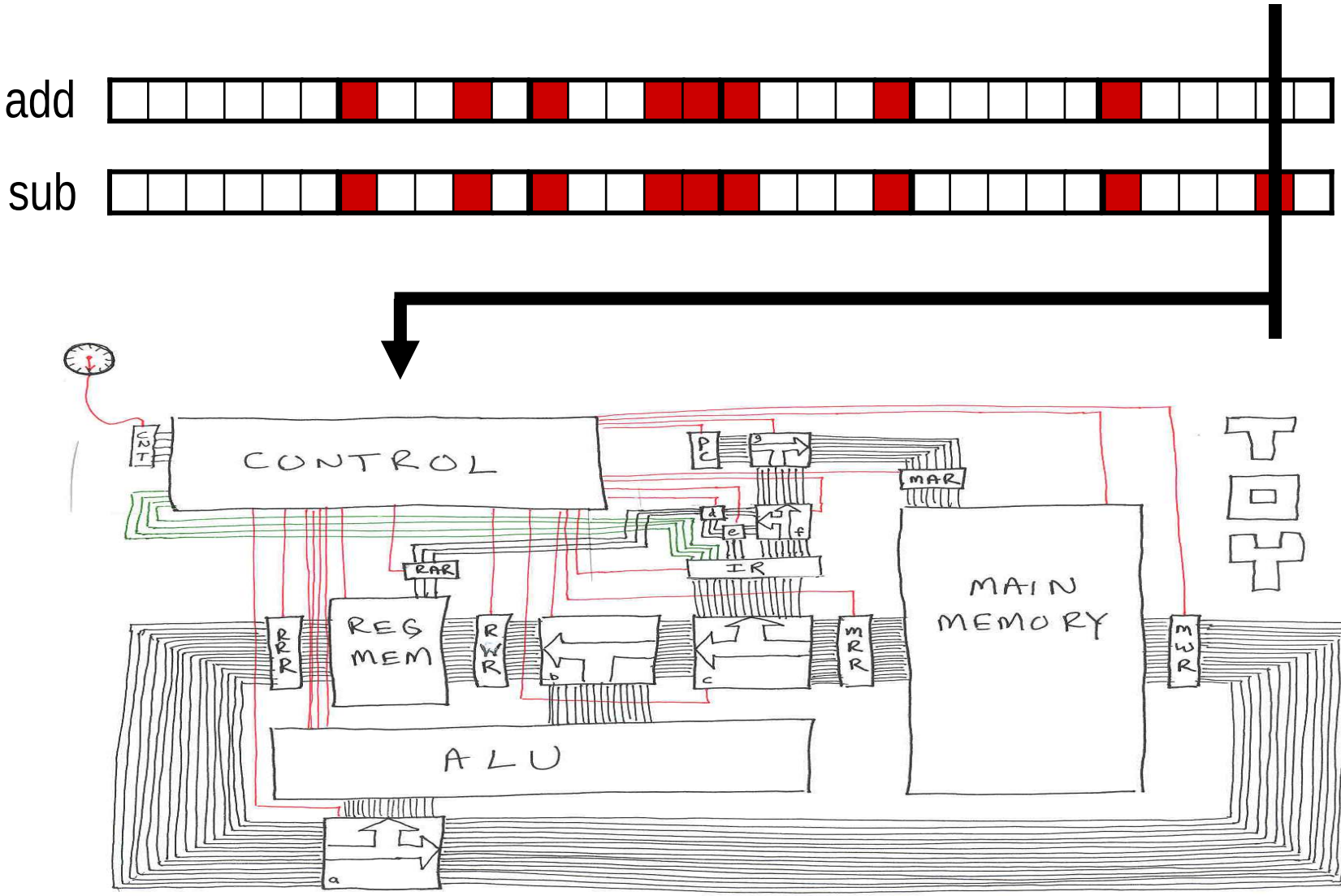
MIPS Add Instruction Encoding



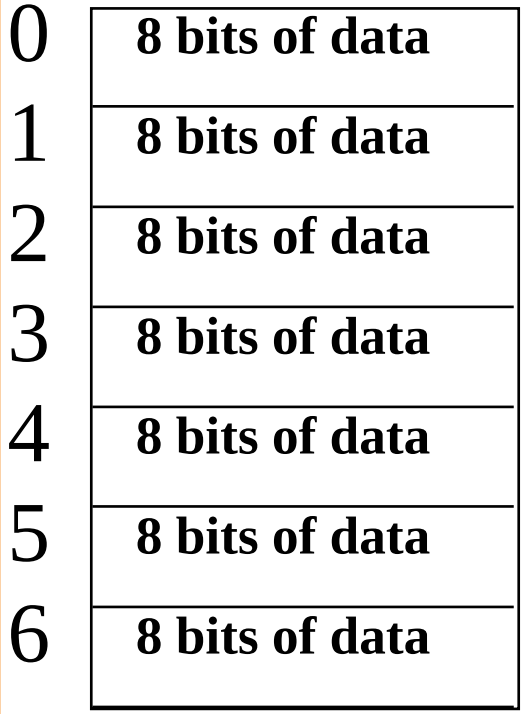
MIPS Add Instruction Encoding



Add and Subtract
A little foreshadowing…



Memory Addressing



View memory as a single-dimensional array

Since 1980: Elements of array are 8-bits

We say “byte addressable”

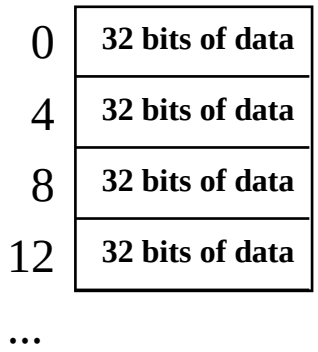
Assuming 32-bit words:

1. How are bytes laid out in word read?

2. Can a word start at any address?

Memory Organization

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



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?

Addressing Modes

Addressing mode	Example	Meaning
Register	Add R4,R3	R4← R4+R3
Immediate	Add R4,#3	R4 ← R4+3
Displacement	Add R4,100(R1)	R4 ← R4+Mem[100+R1]
Register indirect	Add R4,(R1)	R4 ← R4+Mem[R1]
Indexed / Base	Add R3,(R1+R2)	R3 ← R3+Mem[R1+R2]
Direct or absolute	Add R1,(1001)	R1 ← R1+Mem[1001]
Memory indirect	Add R1,@(R3)	R1 ← R1+Mem[Mem[R3]]
Auto-increment	Add R1,(R2)+	R1 ← R1+Mem[R2]; R2 ← R2+d
Auto-decrement	Add R1,–(R2)	R2 ← R2–d; R1 ← R1+Mem[R2]
Scaled	Add R1,100(R2)[R3]	R1 ← R1+Mem[100+R2+R3*d]

Hello World

The Hello World Algorithm:

1. Emit “Hello World”
2. Terminate

C Program

```
/*
 * Good programs have meaningful comments
 */
#include <stdio.h>

int main()
{
  printf("Hello World!\n");
  return 0;
}
```

Hello World

```
/*
 * Good programs have meaningful comments
 */
#include <stdio.h>

int main()
{
  printf("Hello World!\n");
  return 0;
}
```

C Program → GNU C Compiler

GNU C Compiler → IA-64 Assembly Language

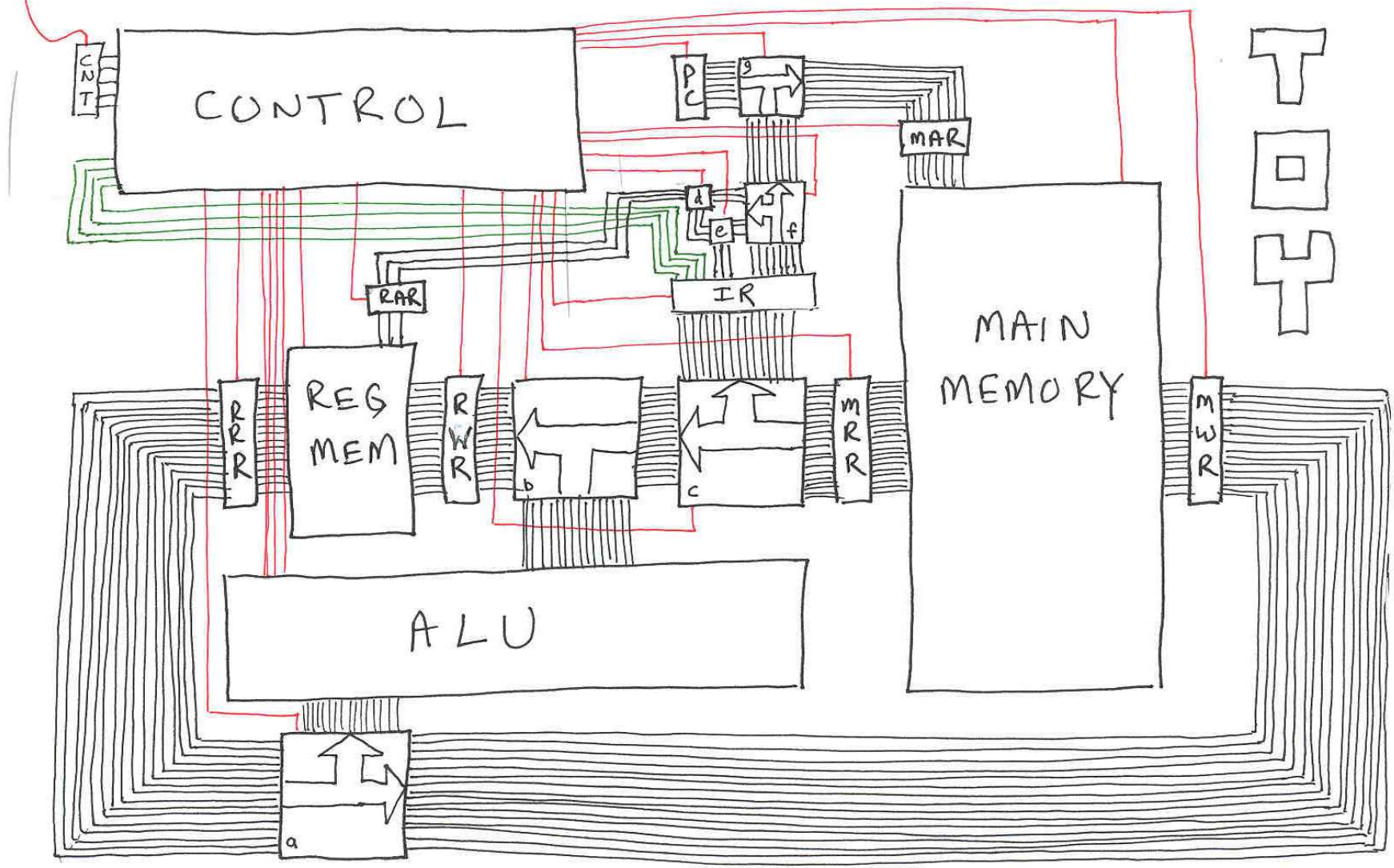
```
        .file   "hello.c"
        .pred.safe_across_calls p1-p5,p16-p63
        .section        .rodata.str1.8,"ams",@progbits,1
        .align 8
.LC0:
        stringz "Hello World!\n"
.text
        .align 16
        .global main#
        .proc main#
main:
        .prologue 12, 33
        .save ar.pfs, r34
        alloc r34 = ar.pfs, 0, 3, 1, 0
        addl r35 = @ltoff(.LC0), gp
        .save rp, r33
        mov r33 = b0
        ;;
        .body
        ld8 r35 = [r35]
        br.call.sptk.many b0 = printf#
        ;;
        mov r8 = r0
        mov ar.pfs = r34
        mov b0 = r33
        br.ret.sptk.many b0
        .endp main#
        .ident  "GCC: (GNU) 2.96 20000731 (Red Hat Linux 7.2 2.96-112.7.2)"
```

Hello World

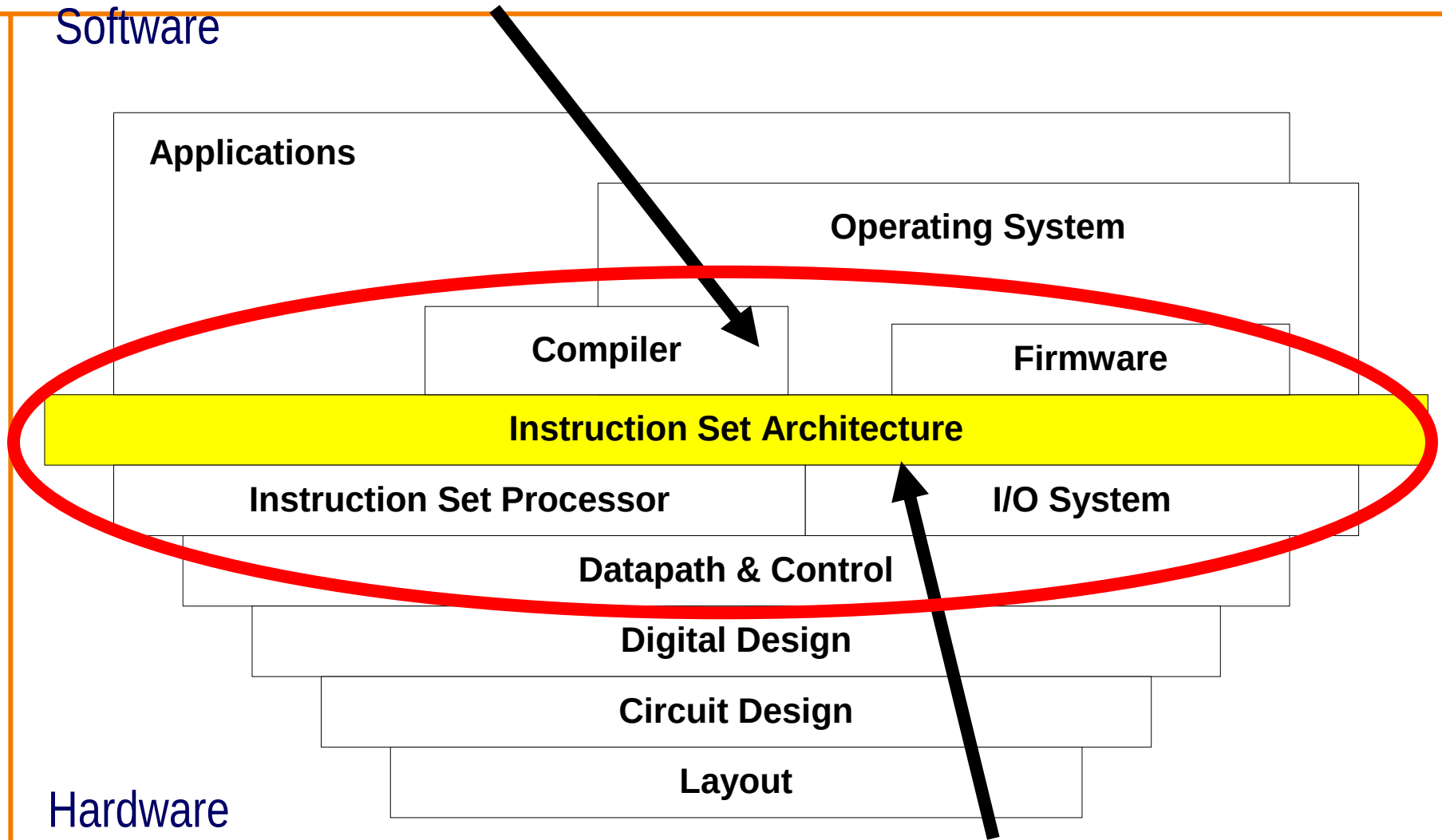
IA-64 Assembly Language

```
        .file   "hello.c"
        .pred.safe_across_calls p1-p5,p16-p63
        .section        .rodata.str1.8,"ams",@progbits,1
        .align 8
.LC0:
        stringz "Hello World!\n"
.text
        .align 16
        .global main#
        .proc main#
main:
        .prologue 12, 33
        .save ar.pfs, r34
        alloc r34 = ar.pfs, 0, 3, 1, 0
        addl r35 = @ltoff(.LC0), gp
        .save rp, r33
        mov r33 = b0
        ;;
        .body
        ld8 r35 = [r35]
        br.call.sptk.many b0 = printf#
        ;;
        mov r8 = r0
        mov ar.pfs = r34
        mov b0 = r33
        br.ret.sptk.many b0
        .endp main#
        .ident  "GCC: (GNU) 2.96 20000731 (Red Hat Linux 7.2 2.96-112.7.2)"
```

Control
(from the back of a napkin)



The Hardware/Software Interface



The Instruction Set Architecture

"The vocabulary of commands"

- Defined by the Architecture (x86)
- Implemented by the Machine (Pentium 4, 3.06 GHz)
- An Abstraction Layer: The Hardware/Software Interface
- Architecture has longevity over implementation
- Example:

add r1 = r2 + r3 (assembly)

001 001 010 011 (binary)

Opcode (verb) Operands (nouns)