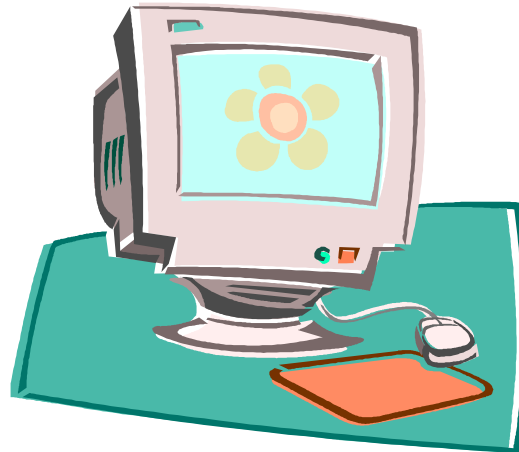


# Other instruction set architectures

---



- Today we'll first look at a longer example program, starting with some C code and translating it into our assembly language from last time.
- Next we discuss some alternative instruction set architectures.
  - There are different ways of specifying memory addresses.
  - Instructions can also have different numbers and types of operands.
- We will use last week's datapath and instruction set for the rest of the lectures and the last machine problem, but it's important to see some other possible designs.

# Representing characters

- All computer data must be stored in a numeric (binary) format—including alphabetic characters!
- Each character can be represented by a one-byte **ASCII code**. The codes for some letters, digits and symbols are shown below.

32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	”	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	\$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	,	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	del

# Representing strings

- In C and C++, a string is stored as an *array* of bytes.
  - Each array element is an ASCII code representing one character.
  - A 0 value marks the end of the array or string.
- For example, “The Godfather” can be stored as a 14-byte array.

84	104	101	32	71	111	100	102	97	116	104	101	114	0
T	h	e		G	o	d	f	a	t	h	e	r	\0

- Array elements are stored in contiguous memory locations. For example, if the first letter of “The Godfather” is at address 1000, the terminating zero will be at address 1013.

1000	1001	1002	1003	1004	1005	1006	1007	1008	1009	1010	1011	1012	1013
84	104	101	32	71	111	100	102	97	116	104	101	114	0
T	h	e		G	o	d	f	a	t	h	e	r	\0

# String manipulation example

---

- Let's write a short program to count the number of space characters that appear in a string.
- We'll start with the higher-level C/C++ code given below.
  - Assume the starting address of the string is stored in variable R0, and the number of spaces found will be placed in variable R3.
  - We just loop over the character array, counting the number of times that ASCII code 32 appears. The loop stops when the array element 0, denoting the end of the string, is found.

```
char R0[];  
int i, R3;  
  
R3 = 0;  
i = 0;  
while (R0[i] != 0) {  
    if (R0[i] == 32)  
        R3++;  
    i++;  
}
```

# Assembly language translation

- Here is a direct translation of the array version.
  - **R0** contains the string's starting address.
  - **R1** contains the loop index, which was "i" in the C code.
  - **R3** contains R0[i].
- We also need the *address* of R0[i] to load the data; this is stored in **R2**.

```
LD    R3, #0           // R3 counts number of spaces
LD    R1, #0           // Use R1 as index i
LOOP: ADD    R2, R0, R1  // R2 = address of R0[i]
LD    R2, (R2)         // R2 = R0[i]
BZ    R2, DONE        // Exit if R2 = 0 (end of string)
SUB   R2, R2, #32     // If R2 != 32 then...
BNZ   R2, NEXT        // ...go on to next character
ADD   R3, R3, #1      // Increment space counter R3
NEXT: ADD   R1, R1, #1 // Increment string index
JMP   LOOP           // Repeat
DONE: ...            // Rest of program
```

# Translation notes

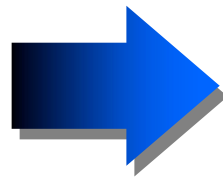
---

- Even though this is such a simple program, it illustrates many differences between high-level and low-level languages!
- Our assembly code uses registers to represent C variables. However, the string itself is stored in RAM, since it could be quite long and we wouldn't necessarily have enough registers.
- The string characters in RAM must be loaded into a register before they can be manipulated.
  - We must specify the exact location of each element that we access. If the RAM word size is one byte and the string starts at address R0, then character R1 will be at address R0 + R1.
  - The C code includes `R0[i]` twice, but we don't have to do two load operations. Instead, we can just re-use register R2.
- Our assembly language has only a limited set of branch instructions, so testing if R2 = 32 takes some extra thought. The code here works since if  $R2 - 32 = 0$ , then  $R2 = 32$ .
- The `DONE` label represents the rest of the program—what's shown here is just a fragment.

## A shorter version

- The loop index **R1** isn't really necessary.
  - We increment R1 on each loop iteration, but all we do is add it to the string address R0, to access the next character of the array.
  - We can make our program a bit shorter by incrementing R0 itself, and dispensing with R1 completely.

```
LD    R3, #0
LD    R1, #0
LOOP: ADD  R2, R0, R1
LD    R2, (R2)
BZ    R2, DONE
SUB   R2, R2, #32
BNZ   R2, NEXT
ADD   R3, R3, #1
NEXT: ADD  R1, R1, #1
      JMP  LOOP
DONE: ...
```



```
LD    R3, #0
LOOP: LD    R2, (R0)
      BZ    R2, DONE
      SUB   R2, R2, #32
      BNZ   R2, NEXT
      ADD   R3, R3, #1
NEXT: ADD  R0, R0, #1
      JMP  LOOP
DONE: ...
```

# Pointers

- This new version corresponds closely to C or C++ code that uses **pointers**, which you may have worked with before.

```
LD    R3, #0
LOOP: LD    R2, (R0)
      BZ    R2, DONE
      SUB  R2, R2, #32
      BNZ  R2, NEXT
      ADD  R3, R3, #1
NEXT:  ADD  R0, R0, #1
      JMP  LOOP
DONE:  ...
```



```
char *R0;
int R3;

R3 = 0;
while (*R0 != 0) {
    if (*R0 == 32)
        R3++;
    R0++;
}
```



# Addressing modes

---

- The first ISA design issue we'll see are different **addressing modes**, which let you specify memory addresses in various ways.
  - Different modes may be useful in different situations.
  - Each mode has its own notation in assembly language.
  - The location that is actually accessed is called the **effective address**.
- The addressing modes that are available will depend on the datapath.
  - Our simple datapath only supports two forms of addressing.
  - Older processors like the 8086 have zillions of addressing modes.
- We'll introduce some of the more common ones.

# Immediate addressing

---

- One of the simplest modes is **immediate addressing**, where the operand itself is accessed.

LD R1, #1999            R1 ← 1999

- This mode is a good way to specify initial values for registers.
- We've already used immediate addressing several times.
  - We introduced it last Wednesday with some short examples.
  - It appears in the string conversion program you just saw.

# Direct addressing

---

- A second mode is **direct addressing**, where the operand is a constant that represents a memory address.

LD R1, 500                      R1 ← M[500]

- Here the effective address is 500, which is just the operand itself.
- This is useful for working with pointers.
  - You can think of the constant as a pointer.
  - The register gets loaded with the data at that address.

# Register indirect addressing

---

- We already saw **register indirect mode**, where the operand is a register that contains a memory address.

`LD R1, (R0)`                       $R1 \leftarrow M[R0]$

- The effective address would be the value in R0.
- This is also useful for working with pointers.
  - R0 might be a pointer, and R1 is loaded with the data at that address.
  - This is similar to `R1 = *R0` in C or C++.
- What's the difference between this register indirect and direct modes?
  - In direct mode, the address is a *constant* that is hard-coded into the program and cannot be changed.
  - Here the contents of R0, and hence the address being accessed, can easily be changed.

# Stepping through arrays

- Register indirect mode makes it easy to access contiguous locations in memory, such as elements of an array.
- If R0 is the address of the first element in an array, we can easily access the second element too.

```
LD R1, (R0)      // R1 contains the first element
ADD R0, R0, #1
LD R2, (R0)      // R2 contains the second element
```

- This is so common that some instruction sets can automatically increment the register for you.

```
LD R1, (R0)+     // R1 contains the first element
LD R2, (R0)+     // R2 contains the second element
```

- Such instructions can be used within loops to access an entire array.

# Indexed addressing

---

- Operands with **indexed addressing** include a constant *and* a register.

LD R1, 500(R0)                      R1 ← M[R0 + 500]

- The effective address is the register data plus the constant. For instance, if R0 contains 25, the effective address here would be 525.
- We can use this addressing mode to access arrays also.
  - The constant is the array address, while the register contains an index into the array.
  - For instance, the instruction above might load the 25th element of an array that starts at memory location 500.
- This form of addressing combines the ideas of direct and register-indirect modes, so it's more complex but also more flexible.
- This is the only mode supported in the MIPS instruction set, which you'll work with more if you go on to take CS232.

# PC-relative addressing

---

- In **PC-relative addressing**, the operand is a constant that is added to the program counter to produce the effective memory address.

`LD R1, $30`                       $R1 \leftarrow M[PC + 30]$

- This is similar to indexed addressing, except the PC is used instead of a regular register.
  - For example, if this instruction was stored at memory address 200, then the effective address would be 230.
  - In some systems the PC actually points to the *next* instruction, in which case the effective address here would be 231.
- Relative addressing is often used in jump and branch instructions.
  - For instance, `JMP $30` lets you skip the next 30 instructions.
  - A negative constant lets you jump backwards, which is common in writing loops as we've seen already.

# Indirect addressing

---

- The most complicated mode that we'll look at is **indirect addressing**. The operand is a constant that represents a memory location, which refers to *another* location, whose contents are then accessed.

LD R1, [360]                      R1 ← M[M[360]]

- The effective address here is M[360].
- Indirect addressing is useful for working with pointers to pointers, which are sometimes also called “handles.”
  - The constant represents a pointer to a pointer.
  - In C, we might write something like `R1 = **ptr`.



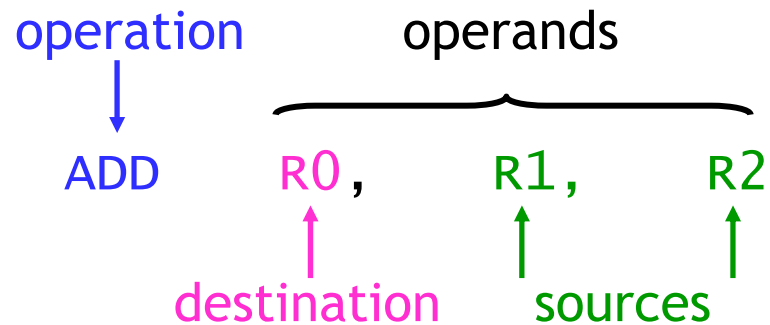
# Addressing mode summary

Mode	Notation	Register transfer equivalent
Immediate	LD R1, #CONST	$R1 \leftarrow \text{CONST}$
Direct	LD R1, CONST	$R1 \leftarrow M[\text{CONST}]$
Register indirect	LD R1, (R0)	$R1 \leftarrow M[R0]$
Indexed	LD R1, CONST(R0)	$R1 \leftarrow M[R0 + \text{CONST}]$
Relative	LD R1, \$CONST	$R1 \leftarrow M[\text{PC} + \text{CONST}]$
Indirect	LD R1, [CONST]	$R1 \leftarrow M[M[\text{CONST}]]$



# Number of operands

- Another way to classify instruction sets is by the number of operands that each data manipulation instruction can have.
- Our examples last week were all **three-address instructions**, because each one had up to three operands—two sources and one destination.



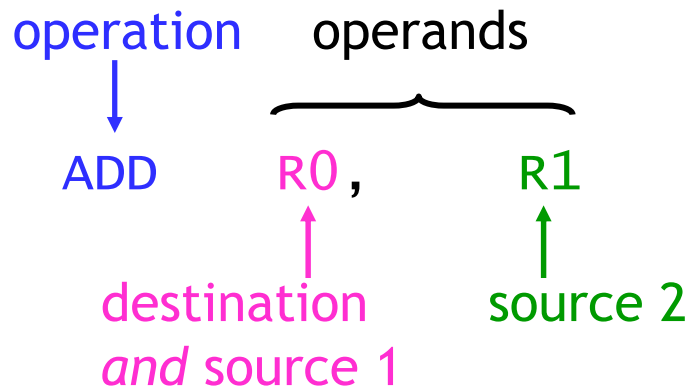
Register transfer instruction:

$R0 \leftarrow R1 + R2$

- This provides the most flexibility, but some instruction sets allow fewer than three operands.

# Two-address instructions

- In **two-address** instructions, the first operand acts as both the destination *and* one of the sources.



Register transfer instruction:

$$R0 \leftarrow R0 + R1$$

- Here are some other examples, with the corresponding register transfer operation and C code.

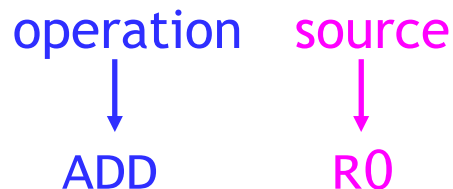
```
ADD R3, #1  
MUL R1, #5  
NOT R1
```

```
R3 ← R3 + 1  
R1 ← R1 * 5  
R1 ← R1'
```

```
R3++;  
R1 *= 5;  
R1 = ~R1;
```

# One-address instructions

- Some computers, like my old Apple II, have **one-address** instructions.
- The CPU has a special register called the **accumulator** which *implicitly* serves as the destination and one of the sources—it “accumulates” the result of a computation.



Register transfer instruction

$$\text{ACC} \leftarrow \text{ACC} + \text{R0}$$

- Here is some code to increment  $M[\text{R0}]$ , using register-indirect addressing.

```
LD   (R0)    ACC ← M[R0]
ADD  #1      ACC ← ACC + 1
ST   (R0)    M[R0] ← ACC
```



# The ultimate: zero addresses

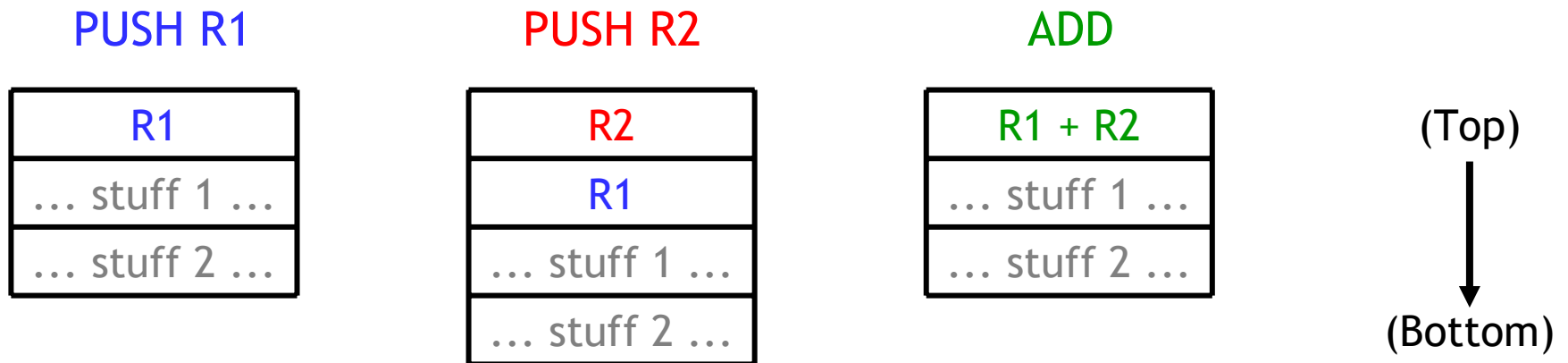
---

- If the destination and sources are *all* implicit, then you wouldn't have to specify any operands at all!
- This is possible with processors that use a **stack architecture**.
  - Operands are **pushed** on a stack, which is just a reserved area in RAM. The most recently pushed element is at the **top of the stack**, or **TOS**.
  - Operations use the topmost stack elements as their operands. Those values are then replaced with the operation's result.
- Examples of stack-based systems include Hewlett-Packard calculators, Java intermediate bytecode, and Intel's 8087 floating-point architecture.



# Stack architecture example

- From left to right, here are three stack instructions, and what the stack looks like after each example instruction is executed.



- This sequence of stack operations corresponds to one register transfer instruction.

$$\text{TOS} \leftarrow R1 + R2$$

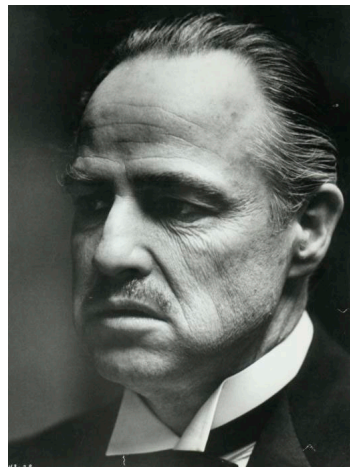
# Data movement instructions

---

- Finally, the *types* of operands allowed in data manipulation instructions is another way of characterizing instruction sets.
  - So far, we've assumed that ALU operations can have only register and constant operands.
  - Many real instruction sets allow memory-based operands as well.
- We'll use the book's example and illustrate how the following operation can be translated into some different assembly languages.

$$X = (A + B) \times (C + D)$$

- Assume that A, B, C, D and X are really memory addresses.



# Register-to-register architectures

- Up until now our programs have used a **register-to-register** or **load/store** architecture, which matches our datapath from last week nicely.
  - Operands in data manipulation instructions must be registers.
  - Data transfer instructions move data between memory and registers.
- In a register-to-register, three-address instruction set, we can translate the operation  $X = (A + B)(C + D)$  into the following code.

```
LD  R1, A      R1 ← M[A]      // Use direct addressing
LD  R2, B      R2 ← M[B]
ADD R3, R1, R2  R3 ← R1 + R2   // R3 = M[A] + M[B]

LD  R1, C      R1 ← M[C]
LD  R2, D      R2 ← M[D]
ADD R1, R1, R2  R1 ← R1 + R2   // R1 = M[C] + M[D]

MUL R1, R1, R3  R1 ← R1 * R3   // R1 has the result
ST  X, R1      M[X] ← R1      // Store that into M[X]
```



# Memory-to-memory architectures

- In a **memory-to-memory** architecture, all data manipulation instructions use memory addresses as operands.
- With a memory-to-memory, three-address instruction set, we could do the operation  $X = (A + B)(C + D)$  a little more simply.

```
ADD X, A, B    M[X] ← M[A] + M[B]
ADD T, C, D    M[T] ← M[C] + M[D]    // Temporary storage
MUL X, X, T    M[X] ← M[X] * M[T]
```

- Here's the same operation but with a *two-address* instruction set.

```
MOVE X, A      M[X] ← M[A]           // Copy M[A] to M[X]
ADD X, B       M[X] ← M[X] + M[B]    // Add M[B]
MOVE T, C      M[T] ← M[C]           // Copy M[C] to M[T]
ADD T, D       M[T] ← M[T] + M[D]    // Add M[D]
MUL X, T       M[X] ← M[X] * M[T]    // Multiply
```

# Register-to-memory architectures

- In a **register-to-memory** architecture, data manipulation instructions can access both registers and memory.
- With two-address instructions, we might do the following.

```
MOV X, A      M[X] ← M[A]           // Copy M[A] to M[X]
ADD X, B      M[X] ← M[X] + M[B]    // Add M[B]
LD  R1, C     R1 ← M[C]             // Load M[C] into R1
ADD R1, D     R1 ← R1 + M[D]        // Add M[D]
MUL X, R1     M[X] ← M[X] * R1     // Multiply
```

# Size and speed

---

- There are many factors to consider when deciding how many and what kind of operands and addressing modes to support in a processor.
- These decisions can affect the **size** of machine language programs.
  - Permitting more operands leads to longer instructions.
  - Memory addresses are long compared to register file addresses (since memories have larger capacities), so instructions with memory-based operands are typically longer than those with register operands.
- There is also an impact on the **speed** of the program.
  - Memory accesses are much slower than register accesses.
  - Longer programs access memory more, just to load the program!
- Most newer processors use register-to-register designs.
  - Reading from registers is faster than reading from RAM.
  - Using register operands also leads to shorter instructions.

# Summary

---

- **Addressing modes** specify how instructions access memory.
  - Our sample ISA supports only **immediate** and **register indirect** modes.
  - Other processors may support many other addressing modes.
- Data manipulation instructions may have from 0 to 3 operands.
  - We will mostly work with a **three-address** instruction set in this class.
  - **Two-address**, **one-address** and **zero-address** instructions are possible.
- Instruction sets may permit different types of operands.
  - Our datapath supports a **register-to-register** architecture, where data operands must be registers or constants.
  - In **register-to-memory** or **memory-to-memory** ISAs, the operands could be memory addresses or a mix of addresses and registers.
- “The Godfather” is a 14-byte C string.