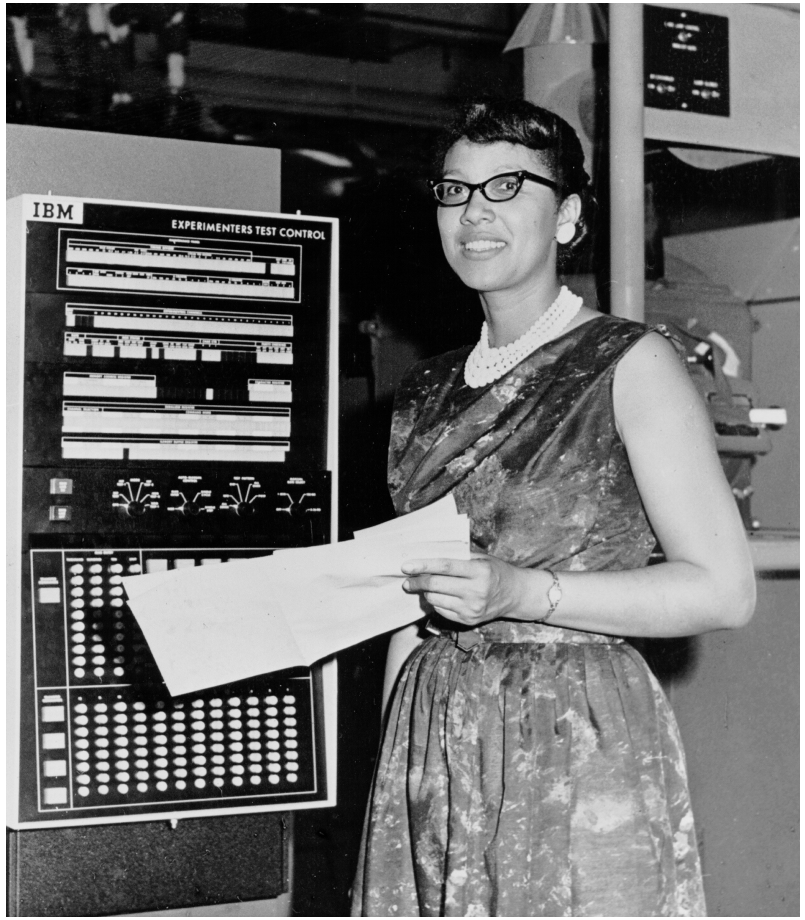
Drawing: Data Representation

- Take three minutes to draw something related to “data representation”

- Some ideas to remember
  - Types (signed, unsigned, char, float, double)
  - Number of bytes
  - Range of values
  - Operations
  - Alignment

Assembly

Dorothy Vaughan, A NASA “Computer”




```
55
48 89 e5
48 83 ec 20
48 8d 05 25 00 00 00
c7 45 fc 00 00 00 00
89 7d f8
48 89 75 f0
48 89 c7
b0 00
e8 00 00 00 00
31 c9
89 45 ec
89 c8
48 83 c4 20
5d
c3
```

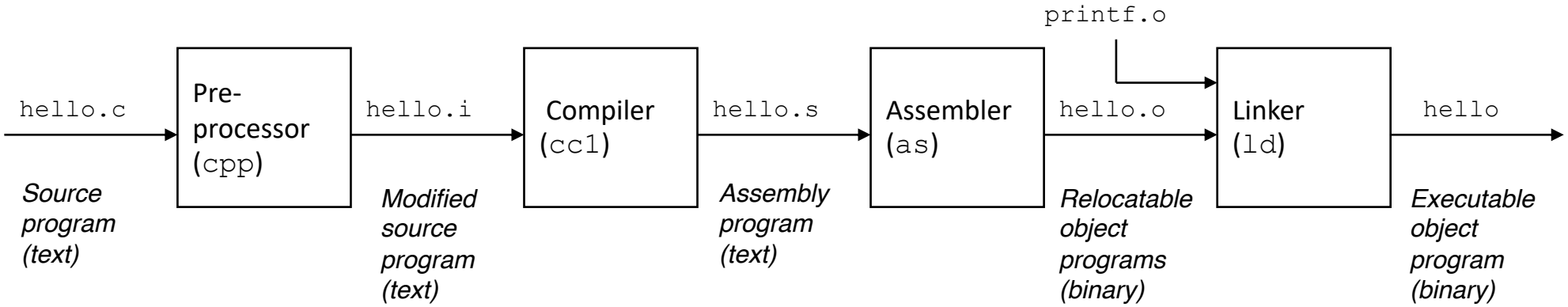
Each ”word” is interpreted by the processor. Different words encode different operations:

- Move some data
- Add two numbers
- Send data to port
- Set interrupt handler

We don’t like thinking in terms of numbers. So, we give them names:

- mov
- add
- out
- int

Compilation Process




```
#include<stdio.h>

int main(int argc,
         char ** argv){

  printf("Hello
          world!\n");
  return 0;
}
```

```
…
int printf(const char *
           restrict,
           ...)
__attribute__((__format_
_ (__printf__, 1, 2)));
…
int main(int argc,
         char ** argv){

  printf("Hello
          world!\n");
  return 0;
}
```

```
pushq   %rbp
movq    %rsp, %rbp
subq  $32, %rsp
leaq  L_.str(%rip), %rax
movl  $0, -4(%rbp)
movl  %edi, -8(%rbp)
movq  %rsi, -16(%rbp)
movq  %rax, %rdi
movb  $0, %al
callq _printf
xorl  %ecx, %ecx
movl  %eax, -20(%rbp)
movl  %ecx, %eax
addq  $32, %rsp
popq  %rbp
retq
```

```
55
48 89 e5
48 83 ec 20
48 8d 05 25 00 00 00
c7 45 fc 00 00 00 00
89 7d f8
48 89 75 f0
48 89 c7
b0 00
e8 00 00 00 00
31 c9
89 45 ec
89 c8
48 83 c4 20
5d
c3
```

Compiler Explorer

Debug Demo

Register	Conventional use	Low 32-bits	Low 16-bits	Low 8-bits	Drawing
rax	**Return** value, callee-owned	eax	ax	al	
rdi	1st **argument**, callee-owned	edi	di	dil	
rsi	2nd **argument**, callee-owned	esi	si	sil	
rdx	3rd **argument**, callee-owned	edx	dx	dl	
rcx	4th **argument**, callee-owned	ecx	cx	cl	
r8	5th **argument**, callee-owned	r8d	r8w	r8b	
r9	6th **argument**, callee-owned	r9d	r9w	r9b	
r10	**Scratch**/temporary, callee-owned	r10d	r10w	r10b	
r11	**Scratch**/temporary, callee-owned	r11d	r11w	r11b	
rsp	**Stack pointer**, caller-owned	esp	sp	spl	
rbx	**Local** variable, caller-owned	ebx	bx	bl	
rbp	**Local** variable, caller-owned	ebp	bp	bpl	
r12	**Local** variable, caller-owned	r12d	r12w	r12b	
r13	**Local** variable, caller-owned	r13d	r13w	r13b	
r14	**Local** variable, caller-owned	r14d	r14w	r14b	
r15	**Local** variable, caller-owned	r15d	r15w	r15b	
rip	**Instruction pointer**				
rflags	**Status**/condition code bits	eflags			

Data Movement Instructions

```
mov DEST, SOURCE ; Equivalent to: dest = source
```

`mov` is an operator (instruction)

`DEST` is an operand (value/data)

`SOURCE` is an operand (value/data)

This is using the (imo, nicer) Intel syntax

There is also the AT&T syntax (e.g., `mov SOURCE, DEST`)

Operands

Form	Syntax	Value	Example
Immediate (binary, octal, decimal, hex, char)	`I`	`I`	`47`
Register	`R`	`Regs[R]`	`rax`
Memory (Absolute)	`[I]`	`Mem[I]`	`[47]`
Memory (Base / Register Indirect)	`[R]`	`Mem[Regs[R]]`	`[rax]`
Memory (Base + Displacement)	`[R + I]`	`Mem[Regs[R] + I]`	`[rbx - 0x2F]`
Memory (Base + Index*Scale + Displacement)	`[R1 + R2*I1 + I2]`	`Mem[Regs[R1] + Regs[R2]*I1 + I2]`	`[rax + rdi*8 + 47]`

Practice with Operands

Register	Value
rax	0x100
rcx	0x001
rdx	0x003

Memory Address	Value
0x100	0x0FF
0x104	0x0AB
0x108	0x013

What are the values of the following operands?

Operand	Description	Answer
rax		
[0x104]		
0x108		
[rax]		
[rax + 4]		

Practice with Operands

Register	Value
rax	0x100
rcx	0x001
rdx	0x003

Memory Address	Value
0x100	0x0FF
0x104	0x0AB
0x108	0x013

What are the values of the following operands?

Operand	Description	Answer
rax	Regs[rax]	0x100
[0x104]	Mem[0x104]	0x0AB
0x108	I	0x108
[rax]	Mem[Regs[rax]]	0x0FF
[rax + 4]	Mem[Regs[rax] + 4]	0x0AB

Operators

Most operators take two operands

- `add R, I`
- `add R, R`
- `add R, M`
- `add M, R`
- `add M, I`

Why not these?

- `add I, R`
- `add I, M`
- `add I, I`
- `add M, M`

`mov` Operator Example

Instruction	Description	C Analog
`mov rax, 0x4`	`rax = 0x4`	`x = 4;`
`mov rda, rax`	`rdx = rax`	`y = i;`
`mov r8 , [0xAB]`	`r8 = Mem[0xAB]`	`z = *p;`
`mov [rax], -147`	`Mem[rax] = -147`	`*p = -147;`
`mov [0xFE], 0x4`	`Mem[0xFE] = 0x4`	`*q = 4;`

Practice with `mov`

For each of the following move instructions, write an equivalent C assignment

1. `mov rbx, 0x40604A`

2. `mov rax, rbx`

3. `mov [rax], 47`

Practice with mov

For each of the following move instructions, write an equivalent C assignment

1. `mov rbx, 0x40604A` `x = 0x40604A;`

2. `mov rax, rbx` `y = x;`

3. `mov [rax], 47` `*y = 47;`

Practice with Translating Assembly

Write a C function `void decode1(long *xp, long *yp)` that will do the same thing as the following assembly code:

```
decode:

        mov rax, [rdi]
        mov rcx, [rsi]
        mov [rsi], rax
        mov [rdi], rcx
        ret
```

```
void decode(long *xp, long *yp){




}
```

Register	Use(s)
rdi	Argument xp
rsi	Argument yp

Practice with Translating Assembly

Write a C function `void decode1(long *xp, long *yp)` that will do the same thing as the following assembly code:

```
decode:

        mov rax, [rdi]
        mov rcx, [rsi]
        mov [rsi], rax
        mov [rdi], rcx
        ret
```

```
void decode(long *xp, long *yp){

    long temp1 = *xp;
    long temp2 = *yp;
    *yp = temp1;
    *xp = temp2;

}
```

Register	Use(s)
rdi	Argument xp
rsi	Argument yp

Some Arithmetic Operations

Operator	Operands	Computation
and	DEST, SRC	DEST = DEST & SRC
or	DEST, SRC	DEST = DEST | SRC
xor	DEST, SRC	DEST = DEST ^ SRC
shl	DEST, SRC	DEST = DEST << SRC
shr	DEST, SRC	DEST = DEST >> SRC
sar	DEST, SRC	DEST = DEST >> SRC
add	DEST, SRC	DEST = DEST + SRC
sub	DEST, SRC	DEST = DEST – SRC
imul	DEST, SRC	DEST = DEST * SRC
not	DEST	DST = ~DEST
inc	DEST	DEST = DEST + 1
dec	DEST	DEST = DEST – 1
neg	DEST	DEST = -DEST

Also called sal

Logical shift right

Arithmetic shift right

Practice with Arithmetic Operators

Register	Value
rax	0x100
rbx	0x108
rdi	0x001

Memory Addr	Value
0x100	0x012
0x108	0x89A
0x110	0x909

Take each instruction independently. The first instruction does not impact the next instruction for this exercise.

Instruction	Description	Result Value	Result Location
add rax, 0x47			
add rax, rbx			
add rax, [rbx]			
add [rax], rbx			
add [rax + rdi*8 + 8], rax			

Practice with Arithmetic Operators

Register	Value
rax	0x100
rbx	0x108
rdi	0x001

Memory Addr	Value
0x100	0x012
0x108	0x89A
0x110	0x909

Take each instruction independently. The first instruction does not impact the next instruction for this exercise.

Instruction	Description	Result Value	Result Location
add rax, 0x47	Reg[rax] += 0x47	0x147	rax
add rax, rbx	Reg[rax] += Reg[rbx]	0x208	rax
add rax, [rbx]	Reg[rax] += Mem[Reg[rbx]]	0x99A	rax
add [rax], rbx	Mem[Reg[rax]] += Reg[rbx]	0x11A	0x100
add [rax + rdi*8 + 8], rax	Mem[Reg[rax] + Reg[rdi]*8 + 8] += Reg[rax]	0xA09	rax

Practice with Translating Assembly

```
arith:

        or      rdi, rsi
        sar     rdi, 3
        not     rdi

        mov     rax, rdx
        sub     rax, rdi

        ret
```

```
long arith(long x, long y, long z){

        x = x | y;
        x = x >> 3;
        x = ~x;

        long return_value = z;
        return_value = return_value - x;

        return return_value;
}
```

Register	Use
rdi	Argument **x**
rsi	Argument **y**
rdx	Argument **z**
rax	return value

C, Assembly, and Object Code

- C Code (store value `t` at location given by pointer `dest`)

```
*dest = t;
```

- Assembly (move value in `rax` to location given by register `rbx`)

```
mov [rbx], rax
```

- Object Code (3-byte instruction found at memory address `0x40059e`)

```
0x40059e:  48 89 03
```