how to asm lol

Table of Contents

Table of Contents

Analytical Engine

- Completely mechanical computer
- Proposed by Charles Babbage in 1837
- Never constructed

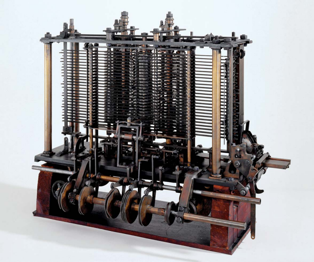


Figure 1: Analytical Engine Part

Register Axles

The core component of the analytical engine

- Three axles each with 50 10-spoked gears
- Each axle represents one 50-digit number

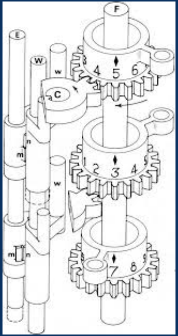


Figure 2: Axle

Register Axles

- Two operand axles A and B
- One egress axle
- Numbers are loaded onto the operand axes
- An operation is performed
- The result is stored onto the egress axis
- $E = A + B$

Register Axles

0	1	2	3	4
0	0	0	0	4

A

+

0	1	2	3	4
0	0	0	0	7

B

0	1	2	3	4
0	0	0	0	0

Egress

Register Axles

0	1	2	3	4
0	0	0	0	4

A

+

0	1	2	3	4
0	0	0	0	7

B

0	1	2	3	4
0	0	0	1	1

Egress

Register Axles

- Modern computers have 16+ digital registers
- 64 bit registers instead of 50 digit decimal registers
- The egress register does not exist on modern computers
- One of the operand registers is overwritten with the result instead

Register Axles

0	1	2	3	4
0	0	0	0	1

A

$+$

0	1	2	3	4
0	0	0	1	1

B

Register Axles

0	1	2	3	4
0	0	1	0	0

A

$+$

0	1	2	3	4
0	0	0	1	1

B

Memory Axles

- Thousands of additional simplified axles
- Two operations
- Load to A/B
- Load from egress
- Each memory axle is labeled with a unique number

Memory Axles

0	1	2	3	4
0	0	1	0	0

Egress

→

0	1	2	3	4
0	0	0	1	1

Memory Axis 100

Figure 2: Store egress into memory axis 100

Memory Axles

0	1	2	3	4
0	0	1	0	0

Egress

$\rightarrow$

0	1	2	3	4
0	0	1	0	0

Memory Axis 100

Figure 2: Store egress into memory axis 100

Memory Axles

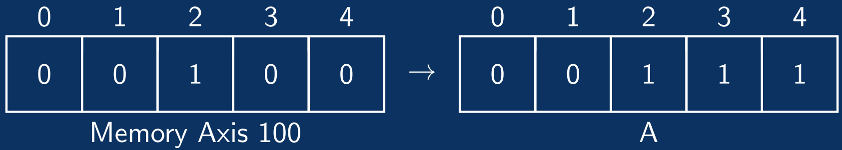


Figure 2: Load Memory Axis 100 to A

Memory Axles

0	1	2	3	4
0	0	1	0	0

Memory Axis 100

$\rightarrow$

0	1	2	3	4
0	0	1	0	0

A

Figure 2: Load Memory Axis 100 to A

Memory Axles

- The memory axles correspond to modern day RAM
- RAM also has unique numbers for each location
- These unique numbers are known as "addresses"

Table of Contents

X86-64

- There are many different "flavors" of assembly
- Each has its own syntax and unique instructions
- This talk uses a simplified form of x86-64 intel syntax

X86-64

- X86-64 has 17 64-bit registers
- Each register has a 32-bit subregister that can be accessed by replacing the r with an e
- Ex: eax is the bottom 32-bits of rax

Register	Size
RAX	8 bytes
RBX	8 bytes
RCX	8 bytes
RDX	8 bytes
RBP	8 bytes
RSI	8 bytes
RDI	8 bytes
RSP	8 bytes
R8	8 bytes
R9	8 bytes
R10	8 bytes
R11	8 bytes
R12	8 bytes
R13	8 bytes
R14	8 bytes
R15	8 bytes
RIP	8 bytes

X86-64

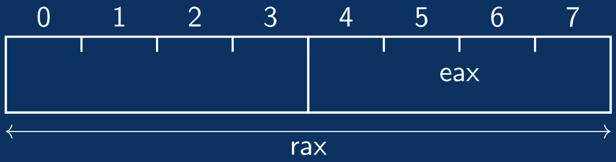


Figure 2: eax

X86-64

- A register in brackets refers to the memory location specified by the register contents

0	1	2	3	4
0	1	1	0	0

rdi

$\rightarrow$

0	1	2	3	4
1	1	1	0	0

Address 0b01100

[rdi] $\rightarrow$ Address

0b01100

Figure 2: [rdi]

Table of Contents

Z3

- First electronic computer
- Konrad Zuse 1941
- Instructions fed on tape

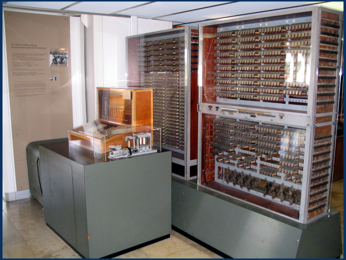


Figure 3: Z3

Computing on the Z3

- More of a programmable calculator than a computer
- Five arithmetic instructions
- Two memory instructions
- Two input/output instructions

Computing on the Z3

Arithmetic Instructions

- add r1, r2 $\rightarrow r1 = r1 + r2$
- sub r1, r2 $\rightarrow r1 = r1 - r2$
- mul r1, r2 $\rightarrow r1 = r1 * r2$
- xor r1, r2 $\rightarrow r1 = r1 \oplus r2$
- and r1, r2 $\rightarrow r1 = r1 \& r2$
- or r1, r2 $\rightarrow r1 = r1 | r2$

Computing on the Z3

Memory instructions

- mov r1, constant $\rightarrow r1 = c$
- mov r1, r2 $\rightarrow r1 = r2$
- mov r1, [r2] $\rightarrow r1 = [r2]$
- mov [r1], r2 $\rightarrow [r1] = r2$

Computing on the Z3

Feed the following tape program into the Z3

```
0: mov rdi, 50
1: mov rsi, 30
2: add rdi, rsi # rdi becomes 80
3: mov rdx, 0
4: mov [rdx], rdi # store 80 at address 0
5: mov rbx, [rdx] # load 80 from address 0 and copy to rbx
```

Table of Contents

Eniac

- Developed by US Army in 1945
- Introduced the concept of a program counter
- Introduced comparisons and conditional jumps

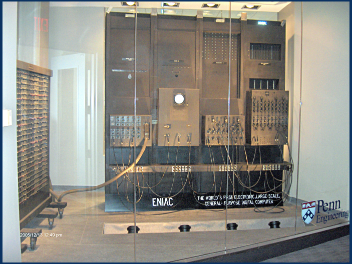
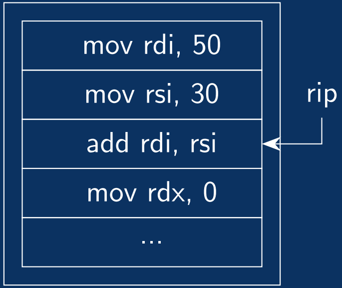


Figure 4: Eniac

Computing on the Eniac

- A special register called rip that indicates what instruction on the tape to execute next
- By modifying rip, we can jump to a new location on the tape

Computing on the Eniac



Tape Program

Computing on the Eniac

- jmp n - changes the program counter to jump to the nth instruction on the tape

Computing on the Eniac

- A special register called FLAGS for storing conditional flags about the last arithmetic operation
- The flags register can be used to make conditional jumps
- Conditional jumps allow for complex structures like if-statements and loops

Computing on the Eniac

0	1	2	3	4
0	0	1	1	1

A

-

0	1	2	3	4
0	0	0	0	1

B

0	1	2	3	4
0	0	1	1	0

Result

0	1	2	3	4
> 0	$\neq 0$		$\dots$	

Flags

Computing on the Eniac

- cmp r1, r2 - sets the conditional flags for r1, r2
- je n - jmp n if the last comparison had r1 $=$ r2
- jne n - jmp n if the last comparison had r1 $\neq$ r2
- jg n - jmp n tape if the last comparison had r1 $>$ r2
- jge n - jmp n if the last comparison had r1 $\geq$ r2
- jl n - jmp n if the last comparison had r1 $<$ r2
- jle n - jmp n if the last comparison had r1 $\leq$ r2

Computing on the Eniac

Now we can make if-statements

```
0: mov rdi, 5 # int x = 5
1: mov rsi, 6 # int y = 6
2: mov rdx, 0 # int z = 0
3: cmp rsi, rdi
4: je 7 # if(x == y) {z = 30}
5: mov rdx, 15 # else {z = 15}
6: jmp 8
7: mov rdx, 30
```

Computing on the Eniac

Now we can make loops

```
0: mov rsi, 0 # int sum = 0
1: mov rdi, 0 # for(int i = 0; i < 10; i++)
2: cmp rdi, 10 #
3: jg 7
4: add rsi, rdi # sum += i
5: add rdi, 1
6: jmp 2
```

Table of Contents

IBM SSEC

- IBM 1948
- Stores instructions in memory
- No more tapes

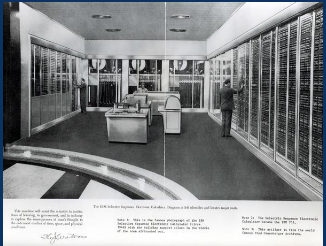
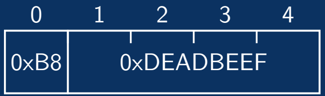


Figure 5: IBM SSEC

Instructions in memory

- An arbitrary scheme is made to encode instructions into numbers
- These numbers can be stored into memory
- The computer can then execute code without a tape



Number Representation

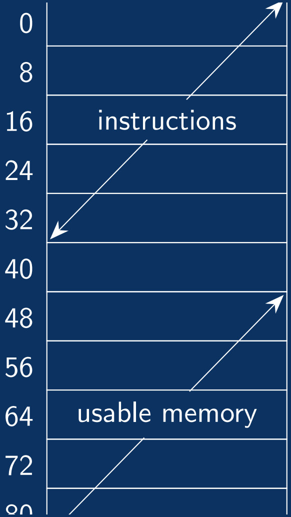
`mov eax, 0xdeadbeef`

Instruction

Instructions in memory

- Memory's address space into two separate, contiguous areas
- One for instructions
- One for the program's memory
- Modern programs divide the address space into many more areas each with a different purpose

Instructions in memory



Instructions in memory

- If we just think of the section for instructions as virtual tape, everything works the same

Table of Contents

EDSAC

- Cambridge 1949
- Introduced a concept called the Wheeler jump
- Allows us to create functions

Figure 6: EDSAC

Computing on the EDSAC

- When calling a function, we need to save the caller address
- Once the function is finished executing, we return to the saved address
- The Wheeler jump is a method for saving and restoring the caller address

Computing on the EDSAC

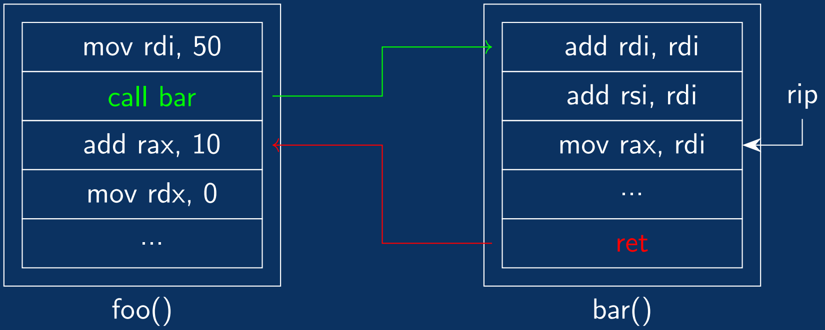


Figure 7: Function call

Computing on the EDSAC

- Function calls natural form a stack
- In modern computers we put these saved addresses into a region called the stack
- When a function is called, the return address is pushed onto the call stack
- When a function returns, the return address is popped and stored into rip
- A special register called rsp was created to keep track of the "top" of the stack

Computing on the EDSAC

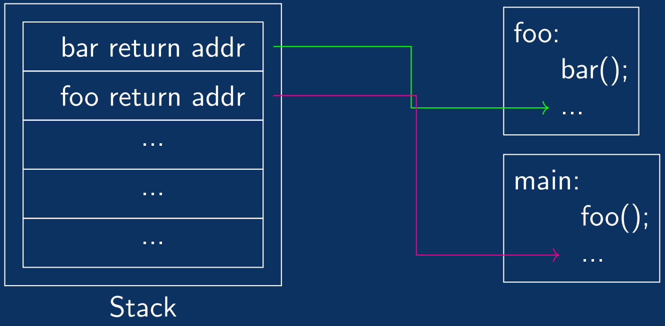


Figure 7: Call Stack

Computing on the EDSAC

- call addr - pushes the return address and jumps to addr
- ret - pop the top of the stack into RIP

Computing on the EDSAC

- The stack also becomes a continuous memory region
- Memory is now partitioned into instructions, general memory and the stack

Computing on the EDSAC

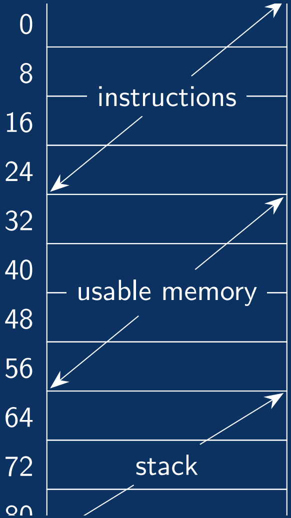


Table of Contents

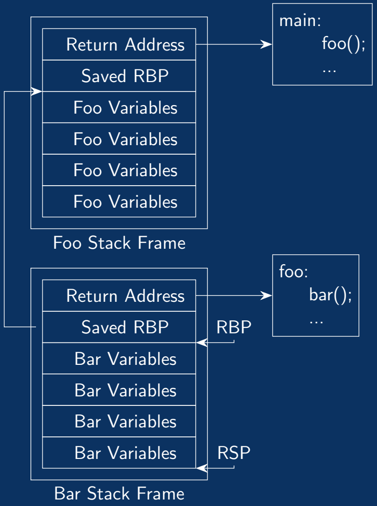
Stack frames

- People realized that the call-stack is a good place to store function local variables
- Another special register was created to keep track of stack frames
- The register rbp points to the bottom of the current stack frame

Stack frames

- To keep track of the base of the previous frame, we put it in the stack frame as well

Stack frames



Stack frames

- push r1 - subtracts 8 from RSP then stores the value of r1 at the new RSP
- pop r1 - loads the value from RSP into r1 then adds 8 to RSP
- leave - the instruction sequence mov rsp, rbp; pop rbp is sometimes shortened to leave
- sub RSP, n - grows the current stack frame by n bytes
- add RSP, n - shrinks the current stack frame by n bytes
- ret - pops into RIP
- call addr - pushes the return address and jumps to addr

Stack frames

```
square(int):
0: push rbp
1: mov rbp, rsp
2: sub rsp, 8 # allocate 8 bytes in stack frame
3: mov [rbp-8], rdi
4: mov rax, DWORD PTR [rbp-8]
5: mul rax, rax
6: pop rbp
7: leave
8: ret
```

Stack frames

```
0: push rbp
1: mov rbp, rsp
2: sub rsp, 8 # allocate 8 bytes in stack frame
3: mov [rbp-8], 0
4: cmp [rbp-8], 10
5: jge 15
6: mov eax, [rbp-8]
7: mov edi, eax
8: call square(int)
9: mov esi, eax
10: mov rdi, 17
11: mov eax, 0
12: call printf
13: add [rbp-8], 1
14: jmp 4
15: leave
16: ret
```

Calling Convention

- Notice that we can arbitrarily pick what registers to pass arguments and return values through
- To make things easier, there is an agreed upon standard called calling convention that specifies these choices

Calling Convention

- The first six arguments are passed through rdi, rsi, rdx, rcx, r8 and r9
- Any successive arguments are pushed onto the stack between stack frames
- The return value for a function is put into rax

Godbolt

- A good way to practice is to write code into godbolt.org and see the assembly output

Questions

Questions?