UiT The Arctic University of Norway

Architecture Crash Course / Task Switching

Project 2 Extra Presentation

Mike Murphy

UiT

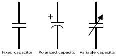
Spring 2025

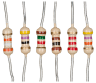
Architecture Crash Course

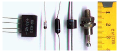
- Some of you have not taken INF-2200, Computer Architecture and Organization.
- That is going to make these assignments difficult,
  because we lean heavily on concepts from that course:
  - Low-level programming
  - Assembly language
  - Registers
  - Opcodes
  - RAM
  - The stack
  - Function calling conventions
- So, here I am going to give you a very quick, very simplified overview.
  - Try to absorb the gist of it.
  - Think of it as a "previously on..."
- Let's start at the lowest level...

Electric Circuits

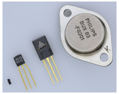
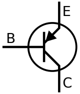
- Voltage, resistance, current. Ohm's Law. $V = I \cdot R$
- Capacitors, resistors, diodes[1]

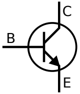








- Transistors[1]

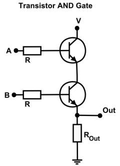


  - Like a valve for current, controlled by voltage
  - E.g. analog amplifier: large current controlled by smaller signal

[1] Images from Michel Bakni, Honina, Omegatron (cc-by-sa), and public domain

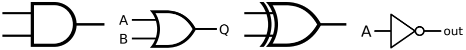
Logic Circuits

- Analog to digital: Voltage to binary: +5V = 1, less = 0
- Combine transistors to build **logic gates**[2]



A	B	A and B
0	0	0
0	1	0
1	0	0
1	1	1

Construction of an AND gate: electric circuit diagram, truth table, logic circuit symbol



Basic logic gates: AND, OR, XOR, NOT

[2] Images from EBatlleP, Helix84 (cc-by-sa), and public domain

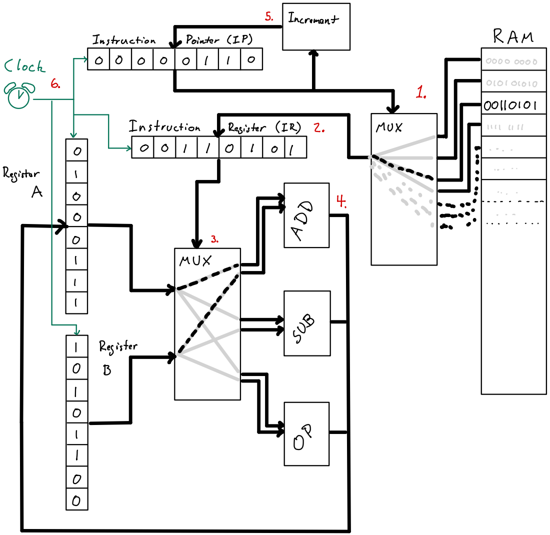
Logic Components

You can combine logic gates to make interesting logic components

- **Latch**: hold one bit of state (1 or 0) until told to update
  - Two inputs: D (data) and E/C (enable/clock)
  - Behavior: When E/C turns to 1, take and hold the value of D (1 or 0)
  - Many latches in parallel → **register**: store a binary number!
- **Adder**: add two binary digits
  - 0 + 1 = binary 01
  - 1 + 1 = binary 10 (0 plus carry)
  - 1 + 1 + previous carry = binary 11 (1 plus carry)
  - Input registers + adders + output register → add binary numbers!
- **Multiplexer**: route signals based on inputs
  - E.g. activate different logic based on value in a register
  - Register + multiplexer + operation circuits to choose from → **instruction**

Basic Computer

Combine registers, adders, multiplexers, and other logic to make a basic computer.



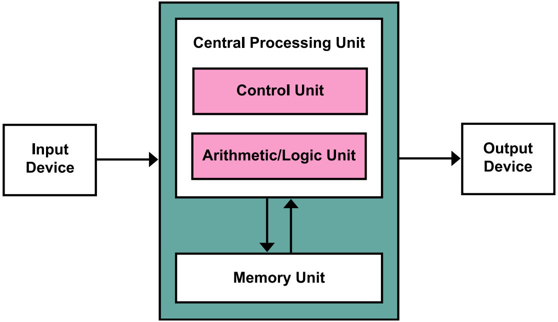
Loop:

1. Instruction Pointer (IP) bits choose RAM slot
2. Instruction loaded to Instruction Register (IR)
3. Instruction bits choose operation (ADD)
4. ADD result fed back to register (A)
5. Instruction Pointer incremented
6. Clock pulse: registers take new values

⟳ Repeat

Von Neumann Architecture

- Dates to WWII and the first programmable computers (ENIAC, EDVAC)
- Named for John von Neumann, who wrote the first published description (1945)
- Basic model for nearly all modern computers[3]



Von Neumann architecture

- **Control Unit**
  Instruction regs, control logic
- **Arithmetic/Logic Unit**
  Data regs, calculation logic
- **Memory (RAM)**
  Short-term storage
- **Input/Output Devices**
  Communication with outside world

[3] Image from Wikipedia user Kapooht (cc-by-sa)

Remember: It's All Bits

- Important to remember: **It's all just bits.**
- The bits activate different circuits that result in different bits.
- In early computers you had to enter bits with switches.[4]

Altair 8800 home computer kit featured in Popular Electronics magazine, 1975

Closer photo of Altair 8800 face with input switches

It's all just bits.
Everything else is an abstraction...

[4] Images from Popular Electronics Magazine, and Todd Dailey (cc-by-sa)

What is an Abstraction?

An abstraction is a way of organizing our thoughts,
so we can ignore details and think of a bigger picture.

We have already seen several of these:

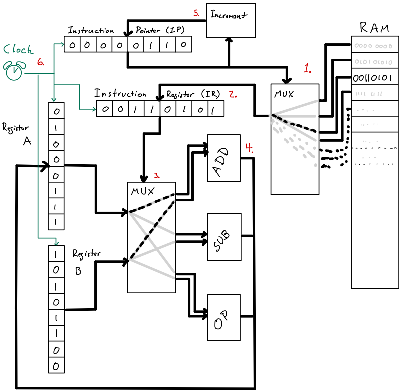
Lower level		Abstraction
analog +5V and 0V	$\rightarrow$	binary 1 and 0
transistor circuit	$\rightarrow$	AND gate
instruction byte 00110101	$\rightarrow$	ADD instruction
sequence of instructions	$\rightarrow$	program

...and so on...

Let's look at some common abstractions...

Machine Language $\rightarrow$ Assembly Language

Recall our simple computer example:



"Step 3. Instruction bits choose operation (ADD)"

- Instruction bits 00110101 are inputs to a multiplexer, these bits select the circuitry for addition.
- This is *machine language:* the actual bits.
- But it's easier to remember `ADD`. This is a *mnemonic.*
- Assembly language:

```
add    %B, %A   # Add A + B, store in A
```

  - Program instruction by instruction
  - But you use mnemonics (ADD, JMP, MOV, etc.) instead of bits

Jump

- **New instruction: JMP**

```
        # ...
        jmp     add_procedure
        # ...

add_procedure:
        add     %B, %A
        # ...
```

- What does a JMP do, in the CPU?

Jump

- **New instruction: JMP**

```
        # ...
        jmp     add_procedure
        # ...

add_procedure:
        add     %B, %A
        # ...
```

- What does a JMP do, in the CPU?
  - Updates the Instruction Pointer (IP) register
  - **IP $\leftarrow$ address of** `add_procedure`

Jump

- **New instruction: JMP**

```
        # ...
        jmp     add_procedure
        # ...

add_procedure:
        add     %B, %A
        # ...
```

- What does a JMP do, in the CPU?
  - Updates the Instruction Pointer (IP) register
  - **IP $\leftarrow$ address of** `add_procedure`
- What is address of `add_procedure`?

Jump

- **New instruction: JMP**

```
        # ...
        jmp     add_procedure
        # ...

add_procedure:
        add     %B, %A
        # ...
```

- What does a JMP do, in the CPU?
  - Updates the Instruction Pointer (IP) register
  - **IP $\leftarrow$ address of `add_procedure`**
- What is address of `add_procedure`?
  - Feature of the assembler: define *symbols*
  - A symbol is a name for a location in the code
  - Later resolved to an actual address by assmbler and linker

Branching: Conditional Jump

- Always jumping is not that useful.
  - We want to check values and make decisions.
  - How do we do that?

Branching: Conditional Jump

- Always jumping is not that useful.
  - We want to check values and make decisions.
  - How do we do that?
- **New control register: FLAGS**
  - Bits set on certain conditions
  - **Z (zero flag)**: set if last result was 0

Branching: Conditional Jump

- Always jumping is not that useful.
  - We want to check values and make decisions.
  - How do we do that?

- **New control register: FLAGS**
  - Bits set on certain conditions
  - **Z (zero flag)**: set if last result was 0

- **New instructions: JZ, JNZ**
  - JZ: Jump if Zero flag set
  - JNZ: Jump if Not Zero

Branching: Conditional Jump

- Always jumping is not that useful.
  - We want to check values and make decisions.
  - How do we do that?

- **New control register: FLAGS**
  - Bits set on certain conditions
  - **Z (zero flag)**: set if last result was 0

- **New instructions: JZ, JNZ**
  - JZ: Jump if Zero flag set
  - JNZ: Jump if Not Zero

```
check_a_eq_1:
        sub     $1, %a      # Subtract 1 from A register (A = A-1).
        jz      a_was_1     # Jump if the Z flag is set (A-1 == 0).
        # Continue here if Z flag was NOT set (A-1 != 0).
        # ...
a_was_1:
        # Jump down here if Z flag was zet (A-1 == 0).
        # ...
```

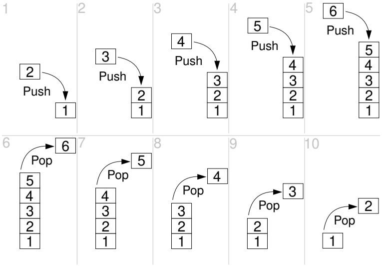
Quick Storage: The Stack

- Data registers are precious
  - Our example has only A and B
  - What if we have a third value in our calculation?
  - What if we need to go do something else?

[5] Image from public domain

Quick Storage: The Stack

- Data registers are precious
  - Our example has only A and B
  - What if we have a third value in our calculation?
  - What if we need to go do something else?
- **New abstraction: Stack**[5]



Stack concept: Last In, First Out (LIFO)

[5] Image from public domain

Stack in a CPU

- **New register: Stack Pointer (SP)**
  - Holds address of last item pushed
- **New instructions: PUSH, POP**
  - **PUSH:** store value on stack
    1. Subtract 1 from SP
       (move to empty slot)
    2. Store value at SP address
  - **POP:** get value off of stack
    1. Load value from SP address
       (last value pushed)
    2. Add 1 to SP
       (move to next value)

```
mov     $0x100, %sp
```

Stack in a CPU

- **New register: Stack Pointer (SP)**
  - Holds address of last item pushed
- **New instructions: PUSH, POP**
  - **PUSH:** store value on stack
    1. Subtract 1 from SP (move to empty slot)
    2. Store value at SP address
  - **POP:** get value off of stack
    1. Load value from SP address (last value pushed)
    2. Add 1 to SP (move to next value)

```
        mov     $0x100, %sp

Regs                    RAM
A  = 0xce      SP ->    | -       | addr 0x100
B  = 0x20               | -       | addr 0x0ff
SP = 0x100              | -       | addr 0x0fe
```

Stack in a CPU

- **New register: Stack Pointer (SP)**
  - Holds address of last item pushed
- **New instructions: PUSH, POP**
  - **PUSH:** store value on stack
    1. Subtract 1 from SP (move to empty slot)
    2. Store value at SP address
  - **POP:** get value off of stack
    1. Load value from SP address (last value pushed)
    2. Add 1 to SP (move to next value)

```
      mov     $0x100, %sp

Regs                    RAM
A  = 0xce     SP ->   | -       | addr 0x100
B  = 0x20             | -       | addr 0x0ff
SP = 0x100            | -       | addr 0x0fe

      push    %a
      mov     $0x01,  %a
```

Stack in a CPU

- **New register: Stack Pointer (SP)**
  - Holds address of last item pushed
- **New instructions: PUSH, POP**
  - **PUSH:** store value on stack
    1. Subtract 1 from SP
       (move to empty slot)
    2. Store value at SP address
  - **POP:** get value off of stack
    1. Load value from SP address
       (last value pushed)
    2. Add 1 to SP
       (move to next value)

```
      mov     $0x100, %sp

Regs                    RAM
A  = 0xce     SP ->   | -      | addr 0x100
B  = 0x20             | -      | addr 0x0ff
SP = 0x100            | -      | addr 0x0fe

      push    %a
      mov     $0x01,  %a

A  = 0x01             | -      | addr 0x100
B  = 0x20     SP ->   | 0xce   | previous A
SP = 0x0ff            | -      | addr 0xfe
```

Stack in a CPU

- **New register: Stack Pointer (SP)**
  - Holds address of last item pushed
- **New instructions: PUSH, POP**
  - **PUSH:** store value on stack
    1. Subtract 1 from SP (move to empty slot)
    2. Store value at SP address
  - **POP:** get value off of stack
    1. Load value from SP address (last value pushed)
    2. Add 1 to SP (move to next value)

```
          mov       $0x100, %sp

Regs                        RAM
A   = 0xce      SP ->     | -       | addr 0x100
B   = 0x20                | -       | addr 0x0ff
SP  = 0x100               | -       | addr 0x0fe

          push      %a
          mov       $0x01,  %a

A   = 0x01                | -       | addr 0x100
B   = 0x20      SP ->     | 0xce    | previous A
SP  = 0x0ff               | -       | addr 0xfe

          push      %b
          mov       $0xff,  %b
```

Stack in a CPU

- **New register: Stack Pointer (SP)**
  - Holds address of last item pushed
- **New instructions: PUSH, POP**
  - **PUSH:** store value on stack
    1. Subtract 1 from SP (move to empty slot)
    2. Store value at SP address
  - **POP:** get value off of stack
    1. Load value from SP address (last value pushed)
    2. Add 1 to SP (move to next value)

```
        mov     $0x100, %sp

Regs                    RAM
A  = 0xce       SP ->   | -       | addr 0x100
B  = 0x20               | -       | addr 0x0ff
SP = 0x100              | -       | addr 0x0fe

        push    %a
        mov     $0x01,  %a

A  = 0x01               | -       | addr 0x100
B  = 0x20       SP ->   | 0xce    | previous A
SP = 0x0ff              | -       | addr 0xfe

        push    %b
        mov     $0xff,  %b

A  = 0x01               | -       | addr 0x100
B  = 0xff               | 0xce    | previous A
SP = 0x0fe      SP ->   | 0x20    | previous B
```

Function Call

- **New abstraction: function**
  - Reusable section of code

Function Call

- **New abstraction: function**
  - Reusable section of code
  - Jump to it, but save place first
  - Return to saved place

Function Call

- **New abstraction: function**
  - Reusable section of code
  - Jump to it, but save place first
  - Return to saved place
- **New instructions: CALL, RET**
  - **CALL:** save place and jump to a function
    1. PUSH IP plus 1 (next instruction)
    2. JMP to given address/symbol
  - **RET:** return to saved place
    1. POP IP

Function Call

- **New abstraction: function**
  - Reusable section of code
  - Jump to it, but save place first
  - Return to saved place
- **New instructions: CALL, RET**
  - **CALL:** save place and jump to a function
    1. PUSH IP plus 1 (next instruction)
    2. JMP to given address/symbol
  - **RET:** return to saved place
    1. POP IP

```
      main:
0x00:        mov     $1, %a
0x01:        mov     $2, %b
0x02:        call    do_add
0x03:        mov     $3, %b
0x04:        call    do_add
0x05:        # ...
      do_add:
0x10:        add     %b, %a
0x11:        ret
```

Function Call

- **New abstraction: function**
  - Reusable section of code
  - Jump to it, but save place first
  - Return to saved place
- **New instructions: CALL, RET**
  - **CALL:** save place and jump to a function
    1. PUSH IP plus 1 (next instruction)
    2. JMP to given address/symbol
  - **RET:** return to saved place
    1. POP IP

```
        main:
0x00:           mov     $1, %a
0x01:           mov     $2, %b
0x02:           call    do_add
0x03:           mov     $3, %b
0x04:           call    do_add
0x05:           # ...
        do_add:
0x10:           add     %b, %a
0x11:           ret
```

1. `IP=0x02 (1st call)  A=1 B=2 Stack:  (empty)`

Function Call

- **New abstraction: function**
  - Reusable section of code
  - Jump to it, but save place first
  - Return to saved place
- **New instructions: CALL, RET**
  - **CALL:** save place and jump to a function
    1. PUSH IP plus 1 (next instruction)
    2. JMP to given address/symbol
  - **RET:** return to saved place
    1. POP IP

```
      main:
0x00:       mov    $1, %a
0x01:       mov    $2, %b
0x02:       call   do_add
0x03:       mov    $3, %b
0x04:       call   do_add
0x05:       # ...
      do_add:
0x10:       add    %b, %a
0x11:       ret
```

1. `IP=0x02 (1st call)  A=1 B=2 Stack:  (empty)`
2. `IP=0x10 (fn:add)    A=1 B=2 Stack:  0x03`

Function Call

- **New abstraction: function**
  - Reusable section of code
  - Jump to it, but save place first
  - Return to saved place
- **New instructions: CALL, RET**
  - **CALL:** save place and jump to a function
    1. PUSH IP plus 1 (next instruction)
    2. JMP to given address/symbol
  - **RET:** return to saved place
    1. POP IP

```
      main:
0x00:        mov     $1, %a
0x01:        mov     $2, %b
0x02:        call    do_add
0x03:        mov     $3, %b
0x04:        call    do_add
0x05:        # ...
      do_add:
0x10:        add     %b, %a
0x11:        ret
```

1. `IP=0x02 (1st call)  A=1 B=2 Stack:  (empty)`
2. `IP=0x10 (fn:add)    A=1 B=2 Stack:  0x03`
3. `IP=0x11 (fn:ret)    A=3 B=2 Stack:  0x03`

Function Call

- **New abstraction: function**
  - Reusable section of code
  - Jump to it, but save place first
  - Return to saved place
- **New instructions: CALL, RET**
  - **CALL:** save place and jump to a function
    1. PUSH IP plus 1 (next instruction)
    2. JMP to given address/symbol
  - **RET:** return to saved place
    1. POP IP

```
      main:
0x00:       mov     $1, %a
0x01:       mov     $2, %b
0x02:       call    do_add
0x03:       mov     $3, %b
0x04:       call    do_add
0x05:       # ...
      do_add:
0x10:       add     %b, %a
0x11:       ret
```

1. IP=0x02 (1st call) A=1 B=2 Stack: (empty)
2. IP=0x10 (fn:add) A=1 B=2 Stack: 0x03
3. IP=0x11 (fn:ret) A=3 B=2 Stack: 0x03
4. IP=0x03 (3rd mov) A=3 B=2 Stack: (empty)

Function Call

- **New abstraction: function**
  - Reusable section of code
  - Jump to it, but save place first
  - Return to saved place
- **New instructions: CALL, RET**
  - **CALL:** save place and jump to a function
    1. PUSH IP plus 1 (next instruction)
    2. JMP to given address/symbol
  - **RET:** return to saved place
    1. POP IP

```
        main:
0x00:           mov     $1, %a
0x01:           mov     $2, %b
0x02:           call    do_add
0x03:           mov     $3, %b
0x04:           call    do_add
0x05:           # ...
        do_add:
0x10:           add     %b, %a
0x11:           ret
```

1. `IP=0x02 (1st call)  A=1 B=2 Stack:  (empty)`
2. `IP=0x10 (fn:add)    A=1 B=2 Stack:  0x03`
3. `IP=0x11 (fn:ret)    A=3 B=2 Stack:  0x03`
4. `IP=0x03 (3rd mov)   A=3 B=2 Stack:  (empty)`
5. `IP=0x04 (2nd call)  A=3 B=3 Stack:  (empty)`

Function Call

- **New abstraction: function**
  - Reusable section of code
  - Jump to it, but save place first
  - Return to saved place
- **New instructions: CALL, RET**
  - **CALL:** save place and jump to a function
    1. PUSH IP plus 1 (next instruction)
    2. JMP to given address/symbol
  - **RET:** return to saved place
    1. POP IP

```
      main:
0x00:         mov     $1, %a
0x01:         mov     $2, %b
0x02:         call    do_add
0x03:         mov     $3, %b
0x04:         call    do_add
0x05:         # ...
      do_add:
0x10:         add     %b, %a
0x11:         ret
```

1. `IP=0x02 (1st call)  A=1 B=2 Stack:  (empty)`
2. `IP=0x10 (fn:add)    A=1 B=2 Stack:  0x03`
3. `IP=0x11 (fn:ret)    A=3 B=2 Stack:  0x03`
4. `IP=0x03 (3rd mov)   A=3 B=2 Stack:  (empty)`
5. `IP=0x04 (2nd call)  A=3 B=3 Stack:  (empty)`
6. `IP=0x10 (fn:add)    A=3 B=3 Stack:  0x05`

Function Call

- **New abstraction: function**
  - Reusable section of code
  - Jump to it, but save place first
  - Return to saved place
- **New instructions: CALL, RET**
  - **CALL:** save place and jump to a function
    1. PUSH IP plus 1 (next instruction)
    2. JMP to given address/symbol
  - **RET:** return to saved place
    1. POP IP

```
    main:
0x00:       mov     $1, %a
0x01:       mov     $2, %b
0x02:       call    do_add
0x03:       mov     $3, %b
0x04:       call    do_add
0x05:       # ...
    do_add:
0x10:       add     %b, %a
0x11:       ret
```

1. IP=0x02 (1st call) A=1 B=2 Stack: (empty)
2. IP=0x10 (fn:add) A=1 B=2 Stack: 0x03
3. IP=0x11 (fn:ret) A=3 B=2 Stack: 0x03
4. IP=0x03 (3rd mov) A=3 B=2 Stack: (empty)
5. IP=0x04 (2nd call) A=3 B=3 Stack: (empty)
6. IP=0x10 (fn:add) A=3 B=3 Stack: 0x05
7. IP=0x11 (fn:ret) A=6 B=3 Stack: 0x05

Function Call

- **New abstraction: function**
  - Reusable section of code
  - Jump to it, but save place first
  - Return to saved place
- **New instructions: CALL, RET**
  - **CALL:** save place and jump to a function
    1. PUSH IP plus 1 (next instruction)
    2. JMP to given address/symbol
  - **RET:** return to saved place
    1. POP IP

```
      main:
0x00:        mov    $1, %a
0x01:        mov    $2, %b
0x02:        call   do_add
0x03:        mov    $3, %b
0x04:        call   do_add
0x05:        # ...
      do_add:
0x10:        add    %b, %a
0x11:        ret
```

```
1. IP=0x02 (1st call)  A=1 B=2 Stack:  (empty)
2. IP=0x10 (fn:add)    A=1 B=2 Stack:  0x03
3. IP=0x11 (fn:ret)    A=3 B=2 Stack:  0x03
4. IP=0x03 (3rd mov)   A=3 B=2 Stack:  (empty)
5. IP=0x04 (2nd call)  A=3 B=3 Stack:  (empty)
6. IP=0x10 (fn:add)    A=3 B=3 Stack:  0x05
7. IP=0x11 (fn:ret)    A=6 B=3 Stack:  0x05
8. IP=0x05 (...)       A=6 B=3 Stack:  (empty)
```

Recap: Basic Example Computer

Example Computer

- Data registers: **A, B**
- Control registers:
  - **IP**: Instruction Pointer
  - **IR**: Instruction Register
  - **FLAGS**: Condition bits
    - **Z flag**: last operation was 0
  - **SP**: Stack Pointer
- Basic loop:
  1. Load IR from IP
  2. IR bits determine logic
  3. Bits propagate though circuits
  4. Clock pulse updates regs
  5. Repeat ↻

Instructions and Abstractions

- Jumping and branching
  - **JMP**: set new IP
  - **JZ, JNZ**: conditional jump
- Stack
  - **PUSH**: put a value on the stack
  - **POP**: take a value off of the stack
- Functions
  - **CALL**: push next IP, then jump
  - **RET**: pop IP to return

x86 Registers

Eight General Purpose Registers: AX, BX, CX, DX, SI, DI, SP, BP

- *General Purpose:* can load/store, do math, etc.
- But may have special talent (like Stack Pointer)
- Named for talent/conventional use

Data

- **AX**: Accumulator
- **DX**: Data
- **BX**: Base

String operations

- **SI**: Source Index
- **DI**: Destination Index
- **CX**: Counter

Stack

- **SP**: Stack Pointer
- **BP**: Base Pointer

Control Registers

- **IP**: Instruction Pointer
- **FLAGS**: Condition bits
- (and more)

Intel Registers: 16-bit to 64-bit

Architecture grew from 16 bits to 32 to 64

- 1978: Intel 8086 is a 16-bit CPU with 16-bit registers
- 1985: Intel 386 expands registers to 32 bits
- 2003: AMD Opteron expands registers to 64 bits

Registers grew and gained prefixes

```
   64|                   32|         16|    8|    0|
     +----+----+----+----+----+----+----+----+
     |                   |         | AH | AL |      H/L = high/low byte
     +----+----+----+----+----+----+----+----+            of 16-bit reg
                                AX |---------|  AX  no prefix = 16 bits
                     EAX |-------------------| EAX   E prefix = 32 bits
 RAX |---------------------------------------| RAX   R prefix = 64 bits
```

- We are working in 32-bit mode, so EAX, ESP, etc.
- But I will not always say the “E” out loud

Function Call Convention

```
/*
 * Example function
 *
 * Lorem ipsum dolor sit amet, consectetur adipiscing elit.
 *
 *  In: SI: format string
 *      AX: first value to format
 *
 *  Out:
 *      AX: number of characters printed
 *
 *  Clobbers: CX, DI
 */
example_print:
    # ...
    ret
```

Not sustainable to have to remember all of this for every function.
So we establish a **calling convention.**

i386 System V Calling Convention

- Parameters are passed on the stack

```
push    %edx      # Param 2
push    %eax      # Param 1
call    my_fn
add     $8, %esp # Remove params
```

i386 System V Calling Convention

- Parameters are passed on the stack

```
push    %edx      # Param 2
push    %eax      # Param 1
call    my_fn
add     $8, %esp # Remove params
```

```
        | ...            |
ESP + 8 | param 2 (EDX)  |
ESP + 4 | param 1 (EAX)  |
ESP ->  | return addr    |
```

i386 System V Calling Convention

- Parameters are passed on the stack

```
push    %edx      # Param 2
push    %eax      # Param 1
call    my_fn
add     $8, %esp  # Remove params
```

```
               | ...             |
ESP + 8 | param 2 (EDX) |
ESP + 4 | param 1 (EAX) |
ESP ->    | return addr     |
```

- Registers AX, CX, and DX are *caller-saved* aka *volatile* aka *scratch*
  - Inside a function, can use without saving
  - When calling a function, assume will be clobbered

i386 System V Calling Convention

- Parameters are passed on the stack

```
push    %edx      # Param 2
push    %eax      # Param 1
call    my_fn
add     $8, %esp # Remove params
```

```
          | ...             |
ESP + 8 | param 2 (EDX) |
ESP + 4 | param 1 (EAX) |
ESP ->  | return addr   |
```

- Registers AX, CX, and DX are *caller-saved* aka *volatile* aka *scratch*
  - Inside a function, can use without saving
  - When calling a function, assume will be clobbered
- Registers BX, SI, DI, SP, BP are *callee-saved* aka *non-volatile*
  - Inside a function, must save/restore if used
  - When calling a function, assume will not be changed

i386 System V Calling Convention

- Parameters are passed on the stack

```
push    %edx     # Param 2
push    %eax     # Param 1
call    my_fn
add     $8, %esp # Remove params
```

```
         | ...            |
ESP + 8  | param 2 (EDX)  |
ESP + 4  | param 1 (EAX)  |
ESP ->   | return addr    |
```

- Registers AX, CX, and DX are *caller-saved* aka *volatile* aka *scratch*
  - Inside a function, can use without saving
  - When calling a function, assume will be clobbered
- Registers BX, SI, DI, SP, BP are *callee-saved* aka *non-volatile*
  - Inside a function, must save/restore if used
  - When calling a function, assume will not be changed
- Return value is passed in AX

Function Stack Frame Pointer

C code

```c
int do_add(int x, int y)
{
    int z = x + y;
    return z;
}
```

Function Stack Frame Pointer

C code

```c
int do_add(int x, int y)
{
    int z = x + y;
    return z;
}
```

Compiled assembly

```
do_add:
    /* Set up stack frame */
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    /* Do addition */
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    addl    %edx, %eax
    /* Save as local z */
    movl    %eax, -4(%ebp)
    /* Return z */
    movl    -4(%ebp), %eax
    /* Undo stack frame */
    leave
    ret
```

Function Stack Frame Pointer

C code

```c
int do_add(int x, int y)
{
    int z = x + y;
    return z;
}
```

Frame Pointer Reg: BP

- BP: “Base Pointer”
- Fixed frame of reference for function
- ESP keeps moving

Compiled assembly

```
do_add:
    /* Set up stack frame */
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    /* Do addition */
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    addl    %edx, %eax
    /* Save as local z */
    movl    %eax, -4(%ebp)
    /* Return z */
    movl    -4(%ebp), %eax
    /* Undo stack frame */
    leave
    ret
```

Function Stack Frame Pointer

C code

```c
int do_add(int x, int y)
{
    int z = x + y;
    return z;
}
```

Frame Pointer Reg: BP

- BP: “Base Pointer”
- Fixed frame of reference for function
- ESP keeps moving

Compiled assembly

```
do_add:
    /* Set up stack frame */
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    /* Do addition */
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    addl    %edx, %eax
    /* Save as local z */
    movl    %eax, -4(%ebp)
    /* Return z */
    movl    -4(%ebp), %eax
    /* Undo stack frame */
    leave
    ret
```

Stack frame

```
    | ...         |
+16 | caller stck |
```

Function Stack Frame Pointer

C code

```c
int do_add(int x, int y)
{
    int z = x + y;
    return z;
}
```

Frame Pointer Reg: BP

- BP: “Base Pointer”
- Fixed frame of reference for function
- ESP keeps moving

Compiled assembly

```
do_add:
    /* Set up stack frame */
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    /* Do addition */
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    addl    %edx, %eax
    /* Save as local z */
    movl    %eax, -4(%ebp)
    /* Return z */
    movl    -4(%ebp), %eax
    /* Undo stack frame */
    leave
    ret
```

Stack frame

```
        | ...          |
    +16 | caller stck  |

    +12 | param y      |
    + 8 | param x      |
```

Function Stack Frame Pointer

C code

```c
int do_add(int x, int y)
{
    int z = x + y;
    return z;
}
```

Frame Pointer Reg: BP

- BP: “Base Pointer”
- Fixed frame of reference for function
- ESP keeps moving

Compiled assembly

```
do_add:
    /* Set up stack frame */
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    /* Do addition */
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    addl    %edx, %eax
    /* Save as local z */
    movl    %eax, -4(%ebp)
    /* Return z */
    movl    -4(%ebp), %eax
    /* Undo stack frame */
    leave
    ret
```

Stack frame

```
    | ...          |
+16 | caller stck  |

+12 | param y      |
+ 8 | param x      |

+ 4 | return addr  |
```

Function Stack Frame Pointer

C code

```c
int do_add(int x, int y)
{
    int z = x + y;
    return z;
}
```

Frame Pointer Reg: BP

- BP: “Base Pointer”
- Fixed frame of reference for function
- ESP keeps moving

Compiled assembly

```
do_add:
    /* Set up stack frame */
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    /* Do addition */
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    addl    %edx, %eax
    /* Save as local z */
    movl    %eax, -4(%ebp)
    /* Return z */
    movl    -4(%ebp), %eax
    /* Undo stack frame */
    leave
    ret
```

Stack frame

```
            | ...           |
        +16 | caller stck   |
        +12 | param y       |
        + 8 | param x       |
        + 4 | return addr   |
EBP ->  + 0 | prev EBP      |
```

Function Stack Frame Pointer

C code

```
int do_add(int x, int y)
{
    int z = x + y;
    return z;
}
```

Frame Pointer Reg: BP

- BP: “Base Pointer”
- Fixed frame of reference for function
- ESP keeps moving

Compiled assembly

```
do_add:
    /* Set up stack frame */
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    /* Do addition */
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    addl    %edx, %eax
    /* Save as local z */
    movl    %eax, -4(%ebp)
    /* Return z */
    movl    -4(%ebp), %eax
    /* Undo stack frame */
    leave
    ret
```

Stack frame

```
              | ...          |
         +16  | caller stck  |

         +12  | param y      |
         + 8  | param x      |

         + 4  | return addr  |

EBP ->   + 0  | prev EBP     |

         - 4  | local var z  |
         - 8  | local ??     |
         -12  | local ??     |
ESP ->   -16  | local ??     |
```

Function Stack Frame Pointer

C code

```c
int do_add(int x, int y)
{
    int z = x + y;
    return z;
}
```

Frame Pointer Reg: BP

- BP: "Base Pointer"
- Fixed frame of reference for function
- ESP keeps moving

Compiled assembly

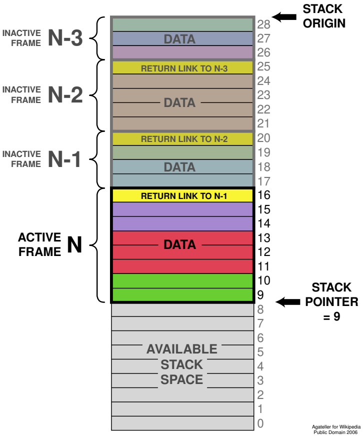
```
do_add:
    /* Set up stack frame */
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    /* Do addition */
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    addl    %edx, %eax
    /* Save as local z */
    movl    %eax, -4(%ebp)
    /* Return z */
    movl    -4(%ebp), %eax
    /* Undo stack frame */
    leave
    ret
```

Stack frame

```
              | ...          |
          +16 | caller stck  |

          +12 | param y      |
          + 8 | param x      |

          + 4 | return addr  |

EBP ->    + 0 | prev EBP     |

          - 4 | local var z  |
          - 8 | local ??     |
          -12 | local ??     |
ESP ->    -16 | local ??     |

          -20 | free stack   |
              | ...          |
```

Stacked Stack Frames

- Nested function calls result in a series of stack frames[6]
- This stack contains the history of how we got here



Series of stack frames from nested calls

Recap: i386 Architecture and Stack Frames

i386

- 8 General Purpose Regs
  - Data: **AX, DX, BX**
  - String Ops: **SI, DI, CX**
  - Stack: **SP, BP**
- Control registers
  - **IP**: Instruction Pointer
  - **FLAGS**: Condition bits
  - and more
- Calling convention
  - Params on stack
  - Caller-saved: AX, CX, DX
  - Callee-saved: BX, SI, DI, SP, BP
  - Return in AX

Stack Frame

```
Prev caller | ...          |    |
            | caller stck | <--\
Caller      +-------------+    |
        + 8 | ..params..  |    |
        + 4 | return addr |    |
        + 0 | prev EBP    | <--\
        - 4 | ..locals..  |    |
Active      +-------------+    |
        + 8 | ..params..  |    |
        + 4 | return addr |    |
EBP ->  + 0 | prev EBP    |----/
ESP ->  - 4 | ..locals..  |
            +-------------+
            | free stack  |
            | ...         |
```

What Does a Process Need to Start?

- Start address

```c
// In `syslib/addrs.h`
#define PROC1_PADDR 0x8000
#define PROC2_PADDR 0xc000
```

- Stack space
  - You need to choose a stack area for the process
  - You can carve up the RAM between 0x20000 and 0x80000

```c
#define T_KSTACK_AREA_MIN_PADDR 0x20000
#define T_KSTACK_AREA_MAX_PADDR 0x80000
#define T_KSTACK_SIZE_EACH      0x1000
#define T_KSTACK_START_OFFSET   0x0ffc
```

How do You Actually Start a Process?

1. In C: set up PCB struct for the process (suggested fn: `createprocess`)
   1.1 Choose stack space and mark it as in use
   1.2 Initialize other fields as needed
2. In assembly: actually start process (Suggested fn: `dispatch`)
   - Have to manipulate registers and stack

   2.1 Move chosen stack value into SP
   2.2 Set initial values for other regs as needed
   2.3 JMP to start address

   - Note that this function won't return normally

```
kernel: dispatch                                  .-> proc_exit(0)
process:        \--> _start --> main --> exit(0) /
```

Process State

- What is the CPU's state?

Process State

- What is the CPU's state?
  - Registers

Process State

- What is the CPU's state?
  - Registers
  - RAM

Process State

- What is the CPU's state?
  - Registers
  - RAM
    - Especially the stack

Process State

- What is the CPU's state?
  - Registers
  - RAM
    - Especially the stack
- What state belongs to the current process?

Process State

- What is the CPU's state?
  - Registers
  - RAM
    - Especially the stack
- What state belongs to the current process?
  - Registers

Process State

- What is the CPU's state?
  - Registers
  - RAM
    - Especially the stack
- What state belongs to the current process?
  - Registers
  - Its data segment, after its code

Process State

- What is the CPU's state?
  - Registers
  - RAM
    - Especially the stack
- What state belongs to the current process?
  - Registers
  - Its data segment, after its code
  - **Its stack**

Saving and Restoring State

- Typically, Instruction Pointer and Stack Pointer move very predictably
  - IP points to code inside function
  - BP points to function's stack pointer
  - SP points to top of stack

  IP and SP move together

  1. IP moves through PUSHes, SP moves down
  2. IP moves through corresponding POPs, SP moves back up
  3. IP moves forward, SP moves down then up symmetrically
  4. RET takes IP back to caller

- To switch tasks, you have to decouple them

  1. IP moves through PUSHes, SP moves down *on caller's stack*
  2. IP moves through code that *switches stacks*
  3. IP moves through corresponding POPs, SP moves back up *on other stack*
  4. RET takes IP back to caller *in other process*

Example Syscall: Write

Flow of control

1. kernel `dispatch`

Stack

1. start of process stack

Example Syscall: Write

Flow of control

1. kernel `dispatch`
2. calls process `_start`

Stack

1. start of process stack
2. frame for `_start`

Example Syscall: Write

Flow of control

1. kernel `dispatch`
2. calls process `_start`
3. calls process `main`

Stack

1. start of process stack
2. frame for `_start`
3. frame for `main`

Example Syscall: Write

Flow of control

1. kernel `dispatch`
2. calls process `_start`
3. calls process `main`
4. calls `write` syscall

Stack

1. start of process stack
2. frame for `_start`
3. frame for `main`
4. frame for `write`

Example Syscall: Write

Flow of control

1. kernel `dispatch`
2. calls process `_start`
3. calls process `main`
4. calls `write` syscall
5. to kernel `dispatch_syscall`

Stack

1. start of process stack
2. frame for `_start`
3. frame for `main`
4. frame for `write`
5. frame for `dispatch_syscall`

Example Syscall: Write

Flow of control

1. kernel `dispatch`
2. calls process `_start`
3. calls process `main`
4. calls `write` syscall
5. to kernel `dispatch_syscall`
6. calls `proc_write`

Stack

1. start of process stack
2. frame for `_start`
3. frame for `main`
4. frame for `write`
5. frame for `dispatch_syscall`
6. frame for `proc_write`

Example Syscall: Write

Flow of control

1. kernel `dispatch`
2. calls process `_start`
3. calls process `main`
4. calls `write` syscall
5. to kernel `dispatch_syscall`
6. calls `proc_write`
7. returns up chain

Stack

1. start of process stack
2. frame for `_start`
3. frame for `main`
4. frame for `write`
5. frame for `dispatch_syscall`
6. frame for `proc_write`

Example Syscall: Write

Flow of control

1. kernel `dispatch`
2. calls process `_start`
3. calls process `main`
4. calls `write` syscall
5. to kernel `dispatch_syscall`

7. returns up chain

Stack

1. start of process stack
2. frame for `_start`
3. frame for `main`
4. frame for `write`
5. frame for `dispatch_syscall`

Example Syscall: Write

Flow of control

1. kernel `dispatch`
2. calls process `_start`
3. calls process `main`
4. calls `write` syscall

7. returns up chain

Stack

1. start of process stack
2. frame for `_start`
3. frame for `main`
4. frame for `write`

Example Syscall: Write

Flow of control

1. kernel `dispatch`
2. calls process `_start`
3. calls process `main`

7. returns up chain

Stack

1. start of process stack
2. frame for `_start`
3. frame for `main`

Example Syscall: Write

Flow of control

1. kernel `dispatch`
2. calls process `_start`
3. calls process `main`

7. returns up chain

Stack

1. start of process stack
2. frame for `_start`
3. frame for `main`

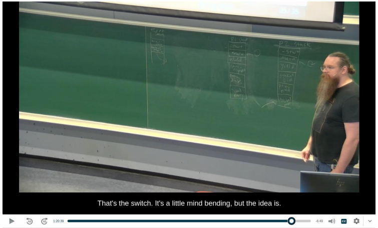
...and `main` continues

Task-Switch Syscall: Yield

At this point I ran out of slides and continued by drawing live on the chalkboard.

The lecture recording is available on Panopto, if you would like to re-watch it.

- Video from beginning
- Jump to this part (1h09m)



Screenshot from lecture recording

The following slides were added after the presentation, with some hints.

(Confusing) Hints in the Precode

- There are remnants of function names and comments in the precode that may seem like hints at a correct solution.
- That solution works like this:

```
yield [in C]
|-- calls scheduler_entry [in asm]
|   |-- saves state of current_running
|   |-- saves stack pointer for current_running             <--- stack save
|   |-- hacks saved stack so that the next return will resume below
|   `-- calls scheduler [in C]
|   .   |-- chooses next process
|   .   |-- sets current_running
|   .   `-- calls dispatch [in ASM]
|   .       |-- loads stack pointer for current running     <-- stack restore
|   .       |-- if the process is running for the first time,
|   .       |       starts new process
|   .       |       process starts running
|   .       |       -- dispatch never returns --
|   .       `-- else
|   .               returns to place saved in scheduler_entry
|   .,-------------'
|   hack_return_point:
|   |-- restores state of new current_running ("on the return path")
|   `-- returns to whatever called scheduler_entry, in this case yield
`-- returns to program
```

- This solution does work, but it is a bit confusing and hard to follow.

A Better Hint

- A tidier solution might look like this:

```
proc_sched_yield [in C]
|-- calls scheduler
|   |-- chooses next process
|   |-- keeps pointer to current process
|   |-- calls switch_task(outgoing, incoming) [in asm]
|   |   |-- saves outgoing process state
|   |   |-- if incoming process is running for the first time
|   |   |       calls dispatch(incoming)
|   |   |       |-- starts new process
|   |   |       `-- process starts running
|   |   |           -- dispatch never returns --
|   |   |-- restores incoming process state
|   |   `-- returns to scheduler
|   `-- returns to whatever called scheduler, in this case proc_sched_yield
`-- returns to program
```

- I like this better because the assembly language is concentrated into the core of the operation, in two functions that do exactly what they say:
  - `switch_task` switches tasks: one task calls, the other returns (or starts)
  - `dispatch` starts a new process
- See this discussion in a thread on Discord

General Advice: Draw, Trace, Debug

- Draw out the contents of your stack at the point where it gets saved
  - What are the last few registers that you pushed?
  - What is the stack pointer (ESP) pointing to when you save it?
  - What is the stack frame pointer (EBP) pointing to?
  - What is the last return address on the stack?

  - When you restore a saved stack, what does it look like?
  - What do you need to pop to restore state?
  - Is the correct return address at the top of the stack when you RET?
- Trace the flow of control (IP) from save to restore
  - If you push more data in this time, what happens to it?
  - What working space do you have in RAM?
  - Can you still use the outgoing stack's stack frame while switching?
- Hint: These drawings would be excellent things to put in your report
- Use your debugger. Step through the switch. Is it doing what you expect?