• **CISC**: Stack-intensive procedure linkage.
  – The stack is used for procedure arguments and return addresses.

• **[Early] RISC**: Register-intensive procedure linkage.
  – Registers are used for procedure arguments and return addresses. Some procedures can thereby avoid any memory references. Typically, the processor has many more (up to 32) registers.
0x100:30f3fcff00000000
4.2 a, p341

- ` irmovl $-4, %ebx`
- `rmmovl %esi, 0x800 (%ebx)`
- `Halt`

- `fcffffff`

ff ff ff fc
int Sum(int *Start, int Count){
    int sum=0;
    while (Count) {
        sum += *Start;
        Start++;
        Count--;
    }
    return sum;
}

Compare to 4.6, p346
pushl %esp
Circuits and HCL

• HCL: Hardware Control Language
  – Describes control logic in processor designs

```c
bool eq = (a && b) ++ (!a && !b);
354, fig 4.10
```

```c
bool Eq = (A == B);
356, fig 4.12
```

```c
bool out = (s && a) ++ (!s && b);
355, fig 4.11
```
ALU: Arithmetic Logic Unit

• Four-way mux:

    int Out4 = [
        !s1 && !s0 : A; #00
        !s1      : B; #01
        !s0      : C; #10
        1        : D; #11
    ];
ALU: Arithmetic Logic Unit

- Four-way mux:
  ```
  int Out4 = [
    !s1 && !s0 : A; #00
    !s1          : B; #01
    !s0          : C; #10
    l            : D; #11
  ];
  ```

- ALU
  
  0: +
  1: -
  2: AND
  3: OR
HCL for circuit that selects min of 3 values?

```c
int Min3 = [
```

```c
];
```
HCL for circuit that selects min of 3 values?

```c
int Min3 = [
    A <= B && A <= C : A;
    B <= A && B <= C : B;
    1 : C;
];
```
Where do Gates Come From?

- Mechanical: Charles Babbage, Analytical Engine
- Vacuum Tubes
Transistors

- \( V_{RC} = I_{CE} \times R_C \), the voltage across the load (the lamp with resistance \( R_C \))
- \( V_{RC} + V_{CE} = V_{CC} \), the supply voltage shown as 6V
Simple Memory

• Cross-coupled NOR gates
  – S-R latch (set, reset latch)
  – Unclocked
  – S = 1 means Q should be 1
  – R = 1 means !Q should be 1
  – S = 1 && R=1?

• D flip-flop (also a D latch)
  – Inputs: data value, D, clock signal, C
  – Outputs: Q, and !Q
  – C = 1 means open, C=0 means closed
Registers: Sequential Circuits

- Combinational Circuit: No internal storage

- Input, State, Output
  - State and output changed on rising edge

- Register File (p. 362)
  - Read ports: srcA/valA, srcB/valB
  - Write port: valW + dstW
  - Clock
  - “val” and “dst” are registers: {0-7, F}
  - Race conditions
Random-Access Memory

- Input: read or write?
- Input: Address, data in
- Output: data out
- Clock
- Error flag
Konrad Zuse


- The z3 (1941) didn't have conditional branching but it did have loops, and it was later proven (technically) Turing complete, and this was 5 years before ENIAC.
Registers: Sequential Circuits

• Combinational Circuit: No internal storage

• Input, State, Output
  – State and output changed on rising edge

• Register File (p. 362)
  – Read ports: srcA/valA, srcB/valB
  – Write port: valW + dstW
  – Clock
  – “val” and “dst” are registers: {0-7, F}
  – Race conditions
Random-Access Memory

- Input: read or write? write
- Input: Address, data in
- Output: data out
- Clock
- Error flag
Intermission

- Michael Winslow
- [http://michaelwinslow.net/](http://michaelwinslow.net/)
- Active: 1980-present
SEQ Stages

- Fetch
  - Read instruction from instruction memory
- Decode
  - Read program registers
- Execute
  - Compute value or address
- Memory
  - Read or write data
- Write Back
  - Write program registers
- PC
  - Update program counter
Sequential Y86 Implementation

1. Fetch
   – From PC
   – icode (4 bits) & ifun (4 bits) [valC (4 bytes)]
   – Calc valP
2. Decode: get rA [rB] [%esp]
3. Execute
   – ALU → valE
   – Conditions set
   – Memory calculations
   – Stack pointer update
4. Memory: valM
5. Write back: registers)
6. PC Update: Set via valP
SEQ Hardware

- **State**
  - Program counter register (PC)
  - Condition code register (CC)
  - Register File
  - Memories
    - Access same memory space
    - Data: for reading/writing program data
    - Instruction: for reading instructions

- **Instruction Flow**
  - Read instruction at address specified by PC
  - Process through stages
  - Update program counter
**Fetch**
- Read 6 bytes

**Decode**
- Read operand registers

**Execute**
- Compute effective address

**Memory**
- Write to memory

**Write back**
- Do nothing

**PC Update**
- Increment PC by 6

<table>
<thead>
<tr>
<th>Step</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>icode:ifun ← M₁[PC]</td>
</tr>
<tr>
<td></td>
<td>rA:rB ← M₁[PC+1]</td>
</tr>
<tr>
<td></td>
<td>valC ← M₄[PC+2]</td>
</tr>
<tr>
<td></td>
<td>valP ← PC+6</td>
</tr>
<tr>
<td>Decode</td>
<td>valA ← R[rA]</td>
</tr>
<tr>
<td></td>
<td>valB ← R[rB]</td>
</tr>
<tr>
<td>Execute</td>
<td>valE ← valB + valC</td>
</tr>
<tr>
<td>Memory</td>
<td>M₄[valE] ← valA</td>
</tr>
<tr>
<td>Write back</td>
<td></td>
</tr>
<tr>
<td>PC update</td>
<td>PC ← valP</td>
</tr>
</tbody>
</table>
rmmovl rA, D(rB)

<table>
<thead>
<tr>
<th>Fetch</th>
<th>icode:ifun ← M1[PC]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>rA:rB ← M1[PC+1]</td>
</tr>
<tr>
<td></td>
<td>valC ← M4[PC+2]</td>
</tr>
<tr>
<td></td>
<td>valP ← PC+6</td>
</tr>
<tr>
<td>Decode</td>
<td>valA ← R[rA]</td>
</tr>
<tr>
<td></td>
<td>valB ← R[rB]</td>
</tr>
<tr>
<td>Execute</td>
<td>valE ← valB + valC</td>
</tr>
<tr>
<td>Memory</td>
<td>M4[valE] ← valA</td>
</tr>
<tr>
<td>Write back</td>
<td></td>
</tr>
<tr>
<td>PC update</td>
<td>PC ← valP</td>
</tr>
</tbody>
</table>