What is a CPU capable of:

- Reading Data
  - from memory or device
- Writing Data
  - to memory or device
- Manipulate Data
  - e.g., add two numbers
- Based on Data decide what to do next
- CPU is an FSM that repeats the following:
  - Get Instruction
  - Read Source Operand Data
  - Perform Calculation
  - Write Result to Destination operand
  - Determine what instruction to execute next

- Instruction:
  - Minimum amount of work the CPU can perform
• Where data comes or goes to?
• Memory:
  – Array of bytes
  – Two things associated with each entry:
    • Address:
      – 32-bit number
    • Value or Contents
      – 8-bit number (byte)
• Two operations:
  – Read from Address - returns byte
  – Write to Address, byte - changes value
• About hex numbers
  – We’ll use 0x?????? Or $?????? For hex numbers
  – Example: 0xabba or $beef1234
  – To convert to binary:
    • Take every digit and convert it into 4 bits
    • Example: 0xabba1234 =
      • 1010 1011 1011 1010 0001 0010 0011 0100
      • a   b   b   a   1   2   3   4
<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000000000</td>
<td>0x02</td>
</tr>
<tr>
<td>0x0000000001</td>
<td>0x30</td>
</tr>
<tr>
<td>0x0000000002</td>
<td>0x04</td>
</tr>
<tr>
<td>0x0000000003</td>
<td>0x05</td>
</tr>
<tr>
<td>0x0000000004</td>
<td>0x61</td>
</tr>
</tbody>
</table>

read 0x0000000000 returns 0x02
read 0x0000000003 returns 0x05
• Can use memory to read/write larger data types

• Data types supported by 68k:
  – Byte
  – Word = 2 bytes
  – Long Word = 4 bytes

• In our previous example
  – Read.w 0x00000000 returns 0x0230
  – Read.l 0x00000000 returns 0x002300405
• Alignment
• A read or write from/to memory address ADDR is aligned if:
  – Accessing a byte: Always
  – Accessing a word: ADDR is divisible by 2 (even)
  – Accessing a long word: ADDR is divisible by 4
• Alignment depends on the datatype accessed and the address
• Big-Endian vs. Little Endian
• When reading words or long words how do we assemble the bytes to form the word or long word
• 68k is Big-Endian: first byte read is the most significant one
  – Read.l 0x00000000 -> 0x02300405
• Some machines are Little-Endian: first byte is the least significant one
  – Read.l 0x00000000 -> 0x05043002
• In addition to source and destination data memory also stores instructions

• How the CPU knows:
  – Where the next instruction is
  – What the instruction is

• Where the next instruction is:
  – Internal 32-bit Register
  – Program Counter or PC
  – Initially points say to address 0x000000000
• What the instruction is
  – Instruction format
    • How to read and interpret memory bytes as instructions
  – This is fixed in the CPU design
• Say we want to do an add.l 0x100, 0x200
  – Mem[0x200] = mem[0x0200] + mem[0x100]
• A possible format is:
  – 1\textsuperscript{st} byte = 0x00 for add or 0x01 for sub
    • 256 different combinations
    • 256 different instructions possible
- Bytes 2, 3, 4 and 5:
  - Source memory address
- Bytes 6, 7, 8, 9
  - Destination memory address
- `add.l $100, $104` becomes:
  - 0x00 -> `add.l`
  - 0x00 -> source memory address = 0x00 00 01 00
  - 0x00
  - 0x10
  - 0x00
  - 0x00
  - 0x00 -> destination memory address = 0x00 00 01 04
  - 0x00
  - 0x01
  - 0x04
• What is the next instruction?
  – For the time being PC is incremented every time we read an instruction byte
  – The next instruction starts where the previous one ended
• This is called straight line sequencing
- `add.l 0x100, 0x104`
- `sub.l 0x104, 0x108`

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00 00 00 00</td>
<td>0x00</td>
<td>add.l</td>
</tr>
<tr>
<td>0x00 00 00 01</td>
<td>0x00</td>
<td>source addr 0x100</td>
</tr>
<tr>
<td>0x00 00 00 02</td>
<td>0x01</td>
<td></td>
</tr>
<tr>
<td>0x00 00 00 03</td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>0x00 00 00 04</td>
<td>0x00</td>
<td>dest addr 0x103</td>
</tr>
<tr>
<td>0x00 00 00 05</td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>0x00 00 00 06</td>
<td>0x01</td>
<td></td>
</tr>
<tr>
<td>0x00 00 00 07</td>
<td>0x04</td>
<td></td>
</tr>
<tr>
<td>0x00 00 00 08</td>
<td>0x01</td>
<td>sub.l</td>
</tr>
<tr>
<td>0x00 00 00 09</td>
<td>0x00</td>
<td>source addr 0x104</td>
</tr>
<tr>
<td>0x00 00 00 0a</td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>0x00 00 00 0b</td>
<td>0x01</td>
<td></td>
</tr>
<tr>
<td>0x00 00 00 0c</td>
<td>0x04</td>
<td></td>
</tr>
<tr>
<td>0x00 00 00 0d</td>
<td>0x00</td>
<td>dest addr 0x108</td>
</tr>
<tr>
<td>0x00 00 00 0e</td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>0x00 00 00 0f</td>
<td>0x01</td>
<td></td>
</tr>
<tr>
<td>0x00 00 00 10</td>
<td>0x08</td>
<td></td>
</tr>
</tbody>
</table>
- **Assume memory is:**
  
<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00 00 01 00</td>
<td>0x00</td>
</tr>
<tr>
<td>0x00 00 01 01</td>
<td>0xff</td>
</tr>
<tr>
<td>0x00 00 01 02</td>
<td>0x00</td>
</tr>
<tr>
<td>0x00 00 01 03</td>
<td>0x01</td>
</tr>
<tr>
<td>0x00 00 01 04</td>
<td>0x01</td>
</tr>
<tr>
<td>0x00 00 01 05</td>
<td>0x00</td>
</tr>
<tr>
<td>0x00 00 01 06</td>
<td>0x00</td>
</tr>
<tr>
<td>0x00 00 01 07</td>
<td>0x02</td>
</tr>
<tr>
<td>0x00 00 01 08</td>
<td>0x01</td>
</tr>
<tr>
<td>0x00 00 01 09</td>
<td>0xff</td>
</tr>
<tr>
<td>0x00 00 01 0a</td>
<td>0xff</td>
</tr>
<tr>
<td>0x00 00 01 0b</td>
<td>0xff</td>
</tr>
</tbody>
</table>
• **add.l 0x100,104**
  - Temp1 = mem[0x100] or 0x00 ff 00 01
  - Temp2 = mem[0x104] or 0x01 00 00 02
  - Temp3 = Temp1 + Temp2 = 0x01 ff 00 03
  - mem[0x104] = 0x01, mem[0x105]=0xff, mem[0x106]=0x00, mem[0x107]=0x03

• **sub.l 0x104, 0x108**
  - Temp1 = mem[0x104], or 0x01 ff 00 03
  - Temp2 = mem[0x108], or 0x01 ff ff ff
  - Temp3 = 0x01ffffff – 0x01ff0003 = 0x0000ffff
  - Mem[0x104]..Mem[107]= 0x0000ffffc
• The problem with Memory
  – Memory is slow
    • 100x the CPU cycle in modern systems
  – Specifying memory addresses for each operand is expensive
    • We need 4 bytes per operand
    • Say we wanted to do add 0x104, 0x100 8 times
    • We need 9 bytes for each of the 8 insts = 72
    • Every instruction reads 2 x 4 bytes and writes 1 x 4 bytes = 12 bytes. 8 instructions = 96 bytes
    • Total = 72 + 96 = 168 bytes 😞
• How to avoid going to memory that often and have short instructions

• Registers
  – Small memory inside CPU
  – Typical size 16 (or 32)
  – Addresses are different from memory addresses
    • Symbolically D0 through D15
  – Each register is 4 bytes long
  – Add now operates on register data
    • E.g, add.l D0, D1 -> D1 = D1 + D0
• To specify an add.l we need
  • 1st byte = 0x00
  • 2nd byte = 0xSD
    – Where S = source register D0-D15
    – And D = destination register D0-D15
  • E.g., 0x00 0x34 = add.l D3, D4

• We now need instructions to move data from registers to memory and vice versa
  – Load addr, Dn    Dn = mem[addr]
  – Store addr, Dn   mem[addr] = Dn
• Format for Load
  – 1\textsuperscript{st} byte = 0x10     -- load.l
  – 2\textsuperscript{nd}-5\textsuperscript{th} byte = addr
  – 6\textsuperscript{th} byte = 0x0N
    – D N = destination register
  – E.g.,
    – 0x10 <- load.l
    – 0x00 < from address 0x00 ff 01 0c
    – 0xff
    – 0x00
    – 0x01
    – 0x0c <- to register D12 ( c )
• **Format for Store**
  
  – 1\textsuperscript{st} byte = 0x11 -- store.l
  
  – 2\textsuperscript{nd}-5\textsuperscript{th} byte = addr
  
  – 6\textsuperscript{th} byte = 0xN0
    
    – D N = source register
  
  – E.g.,
    
    – 0x11 <- store.l
    
    – 0x00 < toaddress 0x00 ff 00 01
    
    – 0xff
    
    – 0x00
    
    – 0x01
    
    – 0x70 <- the contents of register D7 ( c )
• The equivalent of add.l 0x104, 0x100 w/ the new instructions:
  » Load.l 0x100, d1
  » Load.l 0x104, d2
  » Add.l d2, d1
  » Store.l d1, 0x104

• Say I want to add 0x104 to 0x100 8 times:
  – Load.l 0x100, d1 6 bytes, reads 4 bytes
  – Load.l 0x104, d2 6 bytes, reads 4 bytes
  – Add.l d1, d2 2 bytes
  – .repeat 7 times 2 bytes x 7
  – Store d2, 0x104 6 bytes, writes 4 bytes

• Total = 6 x 3 + 2 + 2 x 7 + 4 x 3 = 46 bytes
• Much better than the 168 we needed when everything was in memory
• The 68k CPU
  – 16 Registers
  – D0 through D7
  – A0 through A7
    • D0, D7 can be used with most operations
    • A0-A7 are restricted
    • Usually used to refer to memory
      » Will explain later on
  – All registers 32-bits
    • Can access them as long word (all 32 bits)
    • Word (lower 16 bits)
    • Byte (lower 8 bits)
• Memory in 68k
  – $2^{32}$ bytes
  – Can access as word or long word too
• **A = B + C**
  
  – Where
  
  • A is at memory location $30000$
  • B at $30004$
  • C at $30008$

  • `move.l $30004, d0` \(d0 = \text{mem}[$30004]$\)
  • `move. $30008, d1` \(d1 = \text{mem}[$30008]$\)
  • `add.l d0, d1` \(d1 = d1 + d2\)
  • `move.l d1, $30000` \(\text{mem}[$30000]$ = d1\)
• Writing absolute addresses is cumbersome
• Use Assembly to simplify our life
• You write in a file
  – Compile it with the assembler: a68
  – It produces a binary image which you then load on the machine

    org $300000
    A   dc.l $100
    B   dc.l $200
    C   dc.l $300
    start move.l B, d0
         move.l C, d1
         add.l d0, d1
         move.l d1, A
         trap #15 <- this Ugzmo specific
• Program to sum all the elements of an array w/ 4 elements

    org       $20000
    ARR1    dc.l    $1
    ARR2    dc.l    $2
    ARR3    dc.l    %0000 0100  < binary constant
    ARR4    dc.l    $4
    SUM    dc.l    0    where to place the result

    org       $30000
    move.l    ARR1, d1    first element
    add.l    ARR2, d1
    add.l    ARR3, d1
    add.l    ARR4, d1
    move.l    d1, SUM
    trap #15
- Control flow
- How about:
  - If \((C == 0)\) then \(A = A + B\)
  - else \(A = A - B\)

```
org $20000
C    dc.l  1
A    dc.l  $100
B    dc.l  $200

org $20000
move.l A, d1
move.l C, d0
cmpi.l #0, d0
beq DoADD
DoSUB sub.l B, d1
bra  DONE
DoADD add.l A, d1
DONE move.l d1, A
trap  #15
```
• Branches
  – beq LABEL = Branch Equal
    • If the result of the previous operation was 0
      – PC = LABEL
    • else PC = next instruction PC
  – Bra LABEL = Branch Always
    • Change PC always
    • PC = LABEL

• NOTE:
  – LABEL can only be -32k or +32k away from current PC (it is stored as a relative 16 bit immediate)
  – We will return to branches later on
• How branches work
• One more register:
  – SR = Status Register
    • Bit 0 is the CARRY Condition Code: C
    • Bit 1 is the OVERFLOW CC: V
    • Bit 2 is the ZERO CC: Z
    • Bit 3 is the NEGATIVE CC: N
    • Bit 4 is the EXTENDED CC: X (copy of C)
• Every instruction whose destination is not an A register updates the CCs
• `cmp.i #0, d1`
  – Internally it does `d1 - 0`
  – Assume `d1 = 1`, result = `1 - 0 = 1`
  – `C = 0`
  – `V = 0`
  – `Z = 0`
  – `N = 0`
  – `X = 0`
  – `bne` tests if `Z = 1`
  – `beq` tests if `Z = 0`
• assume d1 = 1
• sub.l #80000000, d1 (-2^31 +1)
  – Z = 1
  – V = 1
  – C = 1
  – N = 0
  – X = 1
• add.l #fffffff, d1
  – Z = 1
  – V = 0
  – C = 1
  – N = 0
  – X = 1
### Table 3-19. Conditional Tests

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Condition</th>
<th>Encoding</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>T*</td>
<td>True</td>
<td>0000</td>
<td>1</td>
</tr>
<tr>
<td>F*</td>
<td>False</td>
<td>0001</td>
<td>0</td>
</tr>
<tr>
<td>HI</td>
<td>High</td>
<td>0010</td>
<td>C A  Z</td>
</tr>
<tr>
<td>LS</td>
<td>Low or Same</td>
<td>0011</td>
<td>C V  Z</td>
</tr>
<tr>
<td>CC(HI)</td>
<td>Carry Clear</td>
<td>0100</td>
<td>C</td>
</tr>
<tr>
<td>CS(LO)</td>
<td>Carry Set</td>
<td>0101</td>
<td>C</td>
</tr>
<tr>
<td>NE</td>
<td>Not Equal</td>
<td>0110</td>
<td>Z</td>
</tr>
<tr>
<td>EQ</td>
<td>Equal</td>
<td>0111</td>
<td>Z</td>
</tr>
<tr>
<td>VC</td>
<td>Overflow Clear</td>
<td>1000</td>
<td>V</td>
</tr>
<tr>
<td>VS</td>
<td>Overflow Set</td>
<td>1001</td>
<td>V</td>
</tr>
<tr>
<td>PL</td>
<td>Plus</td>
<td>1010</td>
<td>N</td>
</tr>
<tr>
<td>MI</td>
<td>Minus</td>
<td>1011</td>
<td>N</td>
</tr>
<tr>
<td>GE</td>
<td>Greater or Equal</td>
<td>1100</td>
<td>N A  V N A  V</td>
</tr>
<tr>
<td>LT</td>
<td>Less Than</td>
<td>1101</td>
<td>N A  V N A  V</td>
</tr>
<tr>
<td>GT</td>
<td>Greater Than</td>
<td>1110</td>
<td>N A  V A  Z V N A  V A  Z</td>
</tr>
<tr>
<td>LE</td>
<td>Less or Equal</td>
<td>1111</td>
<td>Z V N A  V V N A  V</td>
</tr>
</tbody>
</table>

**NOTES:**
- \( \overline{N} \) = Logical Not N
- \( \overline{V} \) = Logical Not V
- \( \overline{Z} \) = Logical Not Z
- *Not available for the Bcc instruction.*
• Adding the elements of an N element array

```
org $20000
N     dc    l 4
ARR   dc    l 1
dc    l 2
dc    l 3
dc    l 4
SUM   dc    l 0

org $30000
start movea    l #ARR,a0 * a0 = $20004  
move    l NUM, d0  d0 = mem[$20000] = 4
clr    l d1     d1 = 0
Loop add    l (a0), d1  d1 = d1 + mem [a0]
          = d1 + mem[$20004]
          = d1 + 1 = 1
add    l #4, a0  a0 = a0 + 4 = $20008
sub    l #1, d0  d1 = d1 - 1
bne    Loop recall if previous result
move    l d1, SUM was NOT zero
```
• `move.l (a0), d1`
  – `Temp1 = read a0`
  – `d1 = mem [temp1]`
• **move.l** $100, d1
  - d1 = mem [$100]
  - Reads from memory
    - C equivalent
      - int d1 = * ((int *) $100);

• **move.l** #$100, d1
  - d1 = $100
  - moves an immediate
    - C equivalent
      - int d1 = 0x100

• **move.l** LABEL, d1
  - d1 = mem[LABEL]

• **move.l** #LABEL, d1
  - d1 = LABEL (pointer)