The Microarchitecture Level

- lies between digital logic level and ISA level
- uses digital circuits to implement machine instructions
- instruction set can be:
  - implemented directly in hardware (RISC)
  - interpreted by microcode (CISC)

The Data Path

- the heart of any CPU
  - ALU, registers, buses

control lines (in)
- 6 ALU: AND, OR, +, ...
- 2 shift:
  - right arithmetic by 1
  - left logical by 8
  - none
- ↑ enable reg onto B bus
- ↑ write C bus to reg

control lines (out)
- Z=1 if result is zero
- N=1 if result is negative

Data Path Ops

- using control lines:
  - 1 operation per clock cycle

Examples:
- arithmetic op with 1 reg and H
  - TOS = MDR + H
- arithmetic with 1 reg
  - H = LV
- shift
  - H = MBR << 8
- same reg can be used on both sides
  - PC = PC + 1
  - due to ALU/shifter delay

memory operations
1. data memory (word-addressed)
2. instruction memory (byte-addressed)

read data memory:
- store address in MAR
- issue rd signal
- MDR holds word after 1 cycle

write data memory:
- store address in MAR
- store word in MDR
- issue wr signal
- memory word contains copy of MDR after 1 cycle
Data Path Ops

- instruction memory fetch:
  - store address in PC
  - issue fetch signal
  - MBR holds byte after 1 cycle
- sign extension (2 controls):
  - choose how to fill rest of MBR
    - unsigned:
      - leading 24 bits = 0
    - signed:
      - leading 24 bits = sign bit
  - E.g.
    - byte=1000 0101 then
    - MBR=1111 1....1 1000 00101

CPU Control

- μ-architecture implements a given instruction set:
  - first define machine instruction set
  - then, for each instruction, provide sets of controls to fetch and execute it
- for RISC:
  - each machine instruction can be hard-coded
   - E.g., assume 3-reg add instruction
   - implement a circuit to:
     - fetch it into IR
     - decode opcode
     - decode each r
     - enable corresponding registers on data path
     - command ALU to add

CPU Control

- for CISC:
  - machine instructions are too complicated
  - each requires a series of signals over multiple clock cycles
  - group signals into microinstructions
    - one executed per clock cycle
  - ALU: 8 bits, as discussed earlier:
    - F0, F1 selects function, ENA/ENB enable A/B, ...
    - SLL8, SRA1 bits control Shifter

- Mem: read, write (data memory), fetch (instr. mem)
Microarchitecture 1

**CPU Control**

- Addr: every μ-instruction specifies its successor
  - unconditional jump, normally to next instruction
  - Addr is modified if JAM bits set \( \iff \) cond. jump
    - JAMZ=1: jump if zero
    - JAMN=1: jump if negative
  - Addr modification for jump: \( \text{NEXT ADDRESS}[8]=1 \)
    - Example: no jump: 092  jump: 192
  - JMPC=1: multiway jump based on opcode

**Data Path with Control**

- MIR (=MBR="IR")
  - control lines for data path
  - memory not shown
  - note decoder for B
- MPC (=PC)
  - next μ-instruction determined by Addr/JAM

**Symbolic μ-Programming**

- writing μ-instructions as bit patterns is too tedious
- use high-level μ-assembly language
- most common operations:
  - \( R1 = H \) op \( R2 \)
    - op: +, AND, OR
    - note: one operand needs to be \( H \)
    - Ex: \( MDR = H + SP \)
    - Effect: set to 1: F0,F1,ENA,ENB,MDR bit in C, SP bit in B
  - \( R1 = R2 \) \( << \) \( 8 \)
    - Ex: \( H = MBR \) \( << \) \( 8 \)
  - \( R = R + 1 \), \( R = R - 1 \)
    - Ex: \( SP = SP - 1 \)
  - \( R1 = R2 = R2 + 1 \)
    - Ex: \( MAR = SP = SP - 1 \)
Symbolic μ-Programming

- rd, wr, fetch
- Ex: MAR = SP; rd
- Note: MDR will contain value at end of next cycle
- goto LABEL
- unconditional jump
- NEXT_ADDRESS is set to LABEL, otherwise to next μ-instr
- if (Z) goto L1; else goto L2
- if (N) goto L1; else goto L2
- sets JAMZ/JAMN bit to implement jump on zero/negative
- goto(MBR)
- multiway jump using current machine opcode

Implementing an ISA

- assume: the target is Java
- local variables for current method (function) kept in a stack frame (LV and SP regs)
- operations are also performed on stack
- Example: a1=a2+a3

```
01234567 |
 SP→  a3
 LV→  a2
 LV→  a1
 SP→  a3+a3
```

- Java source is compiled into byte stream
- each machine instruction:
  - 1-byte opcode
  - followed by 0 or more bytes of operands

Machine Instruction Set

- assume we are given this IS
- Java is compiled into this IS
- each machine instruction is implemented by a sequence of μ-instructions

<table>
<thead>
<tr>
<th>Byte</th>
<th>Mnemonic</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xa1</td>
<td>BRUSH byte</td>
<td>Push byte onto stack</td>
</tr>
<tr>
<td>0x09</td>
<td>DUP</td>
<td>Copy top word on stack and push onto stack</td>
</tr>
<tr>
<td>0x07</td>
<td>JSO</td>
<td>Jump if zero</td>
</tr>
<tr>
<td>0x05</td>
<td>JSR</td>
<td>Jump if sign</td>
</tr>
<tr>
<td>0x01</td>
<td>ILOAD</td>
<td>Push two words from stack push their sum</td>
</tr>
<tr>
<td>0x00</td>
<td>IADD</td>
<td>Push two words from stack push Boolean OR</td>
</tr>
<tr>
<td>0x03</td>
<td>ISTORE</td>
<td>Return from method with integer value</td>
</tr>
<tr>
<td>0x02</td>
<td>IRETURN</td>
<td>Push word from stack and store in local variable</td>
</tr>
<tr>
<td>0x04</td>
<td>IADD</td>
<td>Push two words from stack push their difference</td>
</tr>
<tr>
<td>0x05</td>
<td>ISTORE</td>
<td>Push constant from constant pool onto stack</td>
</tr>
<tr>
<td>0x04</td>
<td>NOP</td>
<td>Do nothing</td>
</tr>
<tr>
<td>0x06</td>
<td>DCP</td>
<td>Denial word on top of stack</td>
</tr>
<tr>
<td>0x0f</td>
<td>DRAK</td>
<td>Swap the two top words on the stack</td>
</tr>
<tr>
<td>0x04</td>
<td>WIDE</td>
<td>Push constant; next instruction has a 64-bit index</td>
</tr>
</tbody>
</table>

Machine Instruction vs μ-Instr.

- Example:
  - Java: a1 = a2 + a3
  - compiler allocates variables in stack frame
  - compiler generates machine code:

<table>
<thead>
<tr>
<th>Byte</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ILOAD 1</td>
</tr>
<tr>
<td>1</td>
<td>ILOAD 2</td>
</tr>
<tr>
<td>2</td>
<td>IADD</td>
</tr>
<tr>
<td>3</td>
<td>ISTORE 0</td>
</tr>
</tbody>
</table>

```
b y t e 0 1 2 3 4 5 6 7 |
 SP→  a3
 LV→  a2
 LV→  a1
 SP→  a3+a3
```

- μ-architecture repeats
  - fetch one byte at a time
  - branch to corresponding sequence of μ-code
  - execute μ-code (fetch operands as appropriate)
  - trace above statement through μ-code
Microprogram

- the machine code for \( a_1 = a_2 + a_3 \) uses the following 3 instructions: iload, istore, iadd

<table>
<thead>
<tr>
<th>Label</th>
<th>Operations</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main1</td>
<td>PC = PC + 1; fetch; goto (MBR)</td>
<td>MBR holds opcode; get next byte; dispatch</td>
</tr>
<tr>
<td>top1</td>
<td>goto Main1</td>
<td>Do nothing</td>
</tr>
<tr>
<td>load1</td>
<td>MAR = SP + SP – 1; rd</td>
<td>Read in next-to-top word on stack</td>
</tr>
<tr>
<td>load2</td>
<td>H = TOS</td>
<td>H = top of stack</td>
</tr>
<tr>
<td>load3</td>
<td>MDR = TOS + MDR + H; wr, goto Main1</td>
<td>Add top two words; write to top of stack</td>
</tr>
<tr>
<td>load4</td>
<td>H = LV</td>
<td>MBR contains index; copy LV to H</td>
</tr>
<tr>
<td>load5</td>
<td>MAR = MBRU + H; rd</td>
<td>MAR = address of local variable to push</td>
</tr>
<tr>
<td>load6</td>
<td>PC = PC + 1; fetch; wr</td>
<td>SP points to next top of stack; prepare write</td>
</tr>
<tr>
<td>load7</td>
<td>TOS = MDR; goto Main1</td>
<td>Inc PC; get next opcode; write top of stack</td>
</tr>
</tbody>
</table>

Microprogram

- every instruction starts with:
  - PC=PC+1, fetch, goto(MBR)
- but: fetch takes 1 cycle
  - MBR contains opcode fetched by previous instruction
  - it fetches the next byte: operand for itself or next opcode
- assume at start: PC=0, MBR=ILOAD

<table>
<thead>
<tr>
<th>byte</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MBR</td>
<td>ILOAD</td>
<td>a2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TOS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Microprogram

- in addition assume:
  - LV, SP point to frame
  - TOS=a3

<table>
<thead>
<tr>
<th>byte</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MBR</td>
<td>ILOAD</td>
<td>a2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TOS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Microprogram

- starting iload

<table>
<thead>
<tr>
<th>byte</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MBR</td>
<td>ILOAD</td>
<td>a2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TOS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Microprogram

- MAR=MBRU+H; rd

<table>
<thead>
<tr>
<th>byte</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MBR</td>
<td>ILOAD</td>
<td>a2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TOS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Microprogram

<table>
<thead>
<tr>
<th>byte</th>
<th>LOAD</th>
<th>ILOAD</th>
<th>ADD</th>
<th>ISTORE</th>
<th>0</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

- **TOS** = MDR; goto Main
- **TOS** = a2
- **MBR** = ILOAD
- **H** = LV

**Starting ILOAD**

### Microprogram

<table>
<thead>
<tr>
<th>byte</th>
<th>LOAD</th>
<th>ILOAD</th>
<th>ADD</th>
<th>ISTORE</th>
<th>0</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

- **TOS** = MDR; goto Main
- **TOS** = a3
- **MBR** = IADD
- **H** = LV

**Starting IADD**
### Microprogram

<table>
<thead>
<tr>
<th>byte</th>
<th>LOAD</th>
<th>ILOAD</th>
<th>IADD</th>
<th>STORE</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ILOAD</td>
<td>ILOAD</td>
<td>IADD</td>
<td>STORE</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

- **MAR=SP+SP-1; rd**: MAR=SP  \( \rightarrow \) a2  
  \( \rightarrow \) a3  
  \( \rightarrow \) a2  
  \( \rightarrow \) a1
- **H=TOS**: H=3  
  MDR=ISTORE  
  H=LV  \( \rightarrow \) a1
- **MDR=TOS=MDR+H; wr; goto Main**: MDR=TOS  \( \rightarrow \) a2+a3
- **PC=PC+1; fetch; goto(MBR)**: PC=6  
  MAR=SP  \( \rightarrow \) a2+a3  
  a3  
  a2  
  a1

### Improving Performance of μ-level

- cost vs speed
- faster clock
- **3-bus architecture**
  - any 2 registers can be used as input to ALU
  - reduces length of μ-code
- separate instruction **fetch unit**
  - next byte is fetched concurrently with execution
  - eliminates Main1: PC=PC+1; fetch; goto(MBR)
  - 1 less cycle for every machine instruction
- **pipelined data path**
Pipelined Data Path

- introduce more parallelism:
  - latches on buses
  - subdivide cycle into μ-steps
  - 4 parallel components:
    1. instruction fetch
    2. registers → input buses
    3. ALU/Shifters
    4. C bus → registers
- instruction is pipelined through the 4 stages
- up to 4 parallel instructions

Pipeline Hazards

- memory read takes one cycle
- instruction may have to wait
- to illustrate, we need a notation for μ-steps
  - ignore fetch (same for every instruction)
  - every μ-instruction can be decomposed into 3 steps
- Consider R1 = R2−1:
  - B=R2 copy R2 to B latch
  - C=B−1 decrement B and place in C latch
  - R1=C copy value from C latch to R1
- Example:
  MAR = SP−1: B=SP; C=B−1; MAR=C

Pipeline Hazards consider the code for swap

- MAR = SP − 1; rd
- MAR = SP
- H = MDR; wr
- MDR = TOS
- MAR = SP − 1; wr
- TOS = H; goto(MBR1)

<table>
<thead>
<tr>
<th>time</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A/B bus</td>
<td>B=H</td>
<td>B=SP</td>
<td>B=TOS</td>
<td>B=MDR</td>
<td>B=SP</td>
<td>B=SP</td>
<td></td>
</tr>
<tr>
<td>ALU</td>
<td>C=B</td>
<td>C=B</td>
<td>C=B</td>
<td>C=B</td>
<td>C=B</td>
<td>C=B</td>
<td></td>
</tr>
<tr>
<td>C bus</td>
<td>MDR=C</td>
<td>MDR=C</td>
<td>H=C;wr</td>
<td>MAR=C</td>
<td>MAR=C;d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A/B bus</td>
<td>B=SP</td>
<td>B=TOS</td>
<td>B=MDR</td>
<td>B=SP</td>
<td>B=SP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALU</td>
<td>C=B</td>
<td>C=B</td>
<td>C=B</td>
<td>C=B</td>
<td>C=B;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C bus</td>
<td>H=C;wr</td>
<td>MAR=C</td>
<td>MAR=C;d</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Cache Memory

- main memory is too slow for the CPU
- use cache:
  - smaller but faster memory
  - holds the most recently accessed portions of memory
  - if word is in cache, no memory access
- based on the locality principle
  - spatial: neighboring words are likely to be accessed
  - temporal: words are accessed repeatedly
Cache Organization

- direct-mapped cache
  - every memory word is mapped to a fixed cache location
  - multiple memory words map to the same location
  
  \[
  \text{main memory:} \quad \begin{array}{c|c|c|c|c}
  | w0 | w1 | \ldots & w0 | w1 | \ldots & w0 | w1 | \ldots & w0 | w1 | \ldots \\
  \hline
  \text{line 0} & \text{line 1} & \text{line k} & \text{line k+1} \\
  \end{array}
  \]

  \[
  \text{cache:} \quad \begin{array}{c|c|c|c|c}
  | w0 | w1 | \ldots & w0 | w1 | \ldots & w0 | w1 | \ldots & w0 | w1 | \ldots \\
  \hline
  \text{line 0} & \text{line 1} \\
  \end{array}
  \]

- each cache line needs a tag to know which memory line it is holding

Cache Organization

- mapping to cache in terms of memory addresses:
  - least significant bits give # bytes/word
  - next least significant bits give # words/line
  - next least significant bits give # lines in cache

  \[
  2^{11} \Rightarrow 2048 \text{ cache lines} \\
  2^5 \Rightarrow 32 \text{ bytes/line} \\
  2^{16} \Rightarrow 65k \text{ mem lines map to same cache line (tag)}
  \]

Cache Organization

- Example: assume 8-bit address
  - 4 B/W
  - 4 W/L
  - 4 L/cache
  - 4 mem lines map to same location

  \[
  \text{TLWB} = \begin{array}{c|c|c|c|c}
  | 00 | 00 | 00 | 00 | \\
  \hline
  | 01 | 10 | 11 |    | \\
  \hline
  | 01 | 00 | 01 | 10 | 11 | \\
  \hline
  | \ldots | \ldots | \ldots | \ldots | \ldots | \\
  | 00 | 01 | 11 | 11 |    | \\
  \hline
  | 00 | 01 | 00 | 00 |    | \\
  \hline
  | 00 | 01 | 11 | 11 |    | \\
  \hline
  | \ldots | \ldots | \ldots | \ldots | \ldots | \\
  | 01 | 00 | 00 | 00 |    | \\
  \hline
  | 01 | 00 | 11 | 11 |    | \\
  \end{array}
  \]

  \[
  \text{line 0} \rightarrow \text{cache line 0} \\
  \text{line 1} \rightarrow \text{cache line 1} \\
  \text{line 4} \rightarrow \text{cache line 0}
  \]

Cache Organization

- direct-mapped cache may lead to “thrashing”
- solution: n-way set-associative cache
  - a word can be in n possible locations
  - less conflict but more management:
    - multiple possible location to check
    - replacement policy, e.g. LRU