What is Control?

- Architecture determines the CPU functionality that is visible to ‘programs’
  - What is the instruction set?
  - What do instructions do?
  - What operations, opcodes?
  - Where are the operands?
  - How to handle interrupts?

- Control Unit, CU (ohjausyksikkö) determines how these things happen in hardware (CPU, MEM, bus, I/O)
  - What gate and circuit should do what at any given time
  - Selects and gives the control signals to circuits in order
  - Physical control wires transmit the control signals
    - Timed by clock pulses
    - Control unit decides values of the signals

---

Functional requirements for CPU
1. Operations
2. Addressing modes
3. Registers
4. I/O module interface
5. Memory module interface
6. Interrupt processing structure
Control Signals

- Main task: control data transfers
  - Inside CPU: REG $\leftrightarrow$ REG, ALU $\leftrightarrow$ REG, ALU-ops
  - CPU $\leftrightarrow$ MEM (I/O-controller): address, data, control
- Timing (ajoitus), Ordering (järjestys)

Micro-Operations

- Simple control signals that cause one very small operation (toiminto)
  - E.g. Bits move from reg 1 through internal bus to ALU
- Subcycle duration determined from the longest operation
- During each subcycle multiple micro-operations in action
  - Some can be done simultaneously,
    - If in different parts of the circuits
  - Must avoid resource conflicts
    - WAR or RAW, ALU, bus
  - Some must be executed sequentially to maintain the semantics

1. $t_1: MAR \leftarrow PC$
2. $t_2: MBR \leftarrow MEM[MAR]$
3. $t_3: IR \leftarrow (MBR)$

If implemented without ALU
Instruction cycle

- When micro-operations address different parts of the hardware, hardware can execute them parallel
- See Chapter 12 instruction cycle examples (next slide)

Instruction Fetch Cycle

Example:
- t1: MAR ← PC
- t2: MAR ← MMU(MAR) → Reserve
- t3: Control Bus ← Read
  - PC ← PC + 1
- t4: MBR ← MEM[MAR] → Release
- t5: IR ← MBR

Execution order? What can be executed parallel? Which micro-ops to same subcycle, which need own cycle?
Instruction Cycle

- Operand fetch cycle(s)
  - From register or from memory
  - Address translation

- Execute cycle(s)
  - Execution often in ALU
  - Operands in and control operation
  - Result from output to register/memoy
  - flags (status)

- Interrupt cycle(s)

**ADD r1, r2, r3:**
- t1: ALUin1 ← r2
- t2: ALUin2 ← r3
- ALUoper ← IR.oper
- t3: r1 ← ALUout
- flags ← xxx

**ISZ X, Increment and Skip if zero:**
- t1: MAR ← IR.address
- t2: MBR ← MEM[MAR]
- t3: MBR ← MBR+1
- t4: MEM[MAR] ← MBR
  - if (MBR=0) then PC ← PC + 1

Instruction Cycle as State-Machine

- ICC: Instruction Cycle Code register’s state

(SysFig 15.3)
Instruction Cycle Control as State-Machine

- Functionality of Control Unit can be presented as state-machine
  - State: What stage of the instruction cycle is going on in CPU
  - Substate: timing based, group of micro-operations executed parallel in one (sub)cycle

- Substate control signals are based on
  - (sub)state itself
  - Fields of IR-register (opcode, operands)
  - Previous results (flags)
    = Execution

- New state based on previous state and flags
  - Also external interrupts effect the new state = Sequencing

Control signals

- Micro-operation ⇒ CU emits a set of control signals
- Example: processor with single accumulator
Control Signals and Micro-Operations

<table>
<thead>
<tr>
<th>Micro-operations</th>
<th>Timing</th>
<th>Active Control Signals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t₀: MAR ← (PC)</td>
<td>C₄</td>
<td></td>
</tr>
<tr>
<td>t₁: MBR ← Memory</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t₂: IR ← (PC+1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Indirect</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t₀: MAR ← (IR(Address))</td>
<td>C₃</td>
<td></td>
</tr>
<tr>
<td>t₁: MBR ← Memory</td>
<td>C₃, R</td>
<td></td>
</tr>
<tr>
<td>t₂: IR(Address) ← (MBR(Address))</td>
<td>C₄</td>
<td></td>
</tr>
<tr>
<td>Interrupt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t₀: MBR ← (PC)</td>
<td>C₁</td>
<td></td>
</tr>
<tr>
<td>t₁: MAR ← (IR(Err))</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t₂: MAR ← same address</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t₃: Memory ← (MBR)</td>
<td>C₁₀, W</td>
<td></td>
</tr>
</tbody>
</table>

C₃ = Read control signal to system bus.
C₁₀ = Write control signal to system bus.

Internal Processor Organization

- Fig 15.5 too complex wiring for implementation?
- Use internal processor bus to connect the components
- ALU usually has temporary registers Y and Z

ADD I:
- t₁: MAR ← IR.address
- t₂: MBR ← MEM[MAR]
- t₃: Y ← MBR
- t₄: Z ← AC + Y
- t₅: AC ← Z
Hardwired implementation
*(Langoitettu ohjaus)*

- Can be used when CU’s inputs and outputs fixed
  - Functionality described using Boolean logic
  - CU implemented by one logical circuit
  - Example: $C_5 = P^*Q*T_2 + P^*Q^*(LDA)^*T_2 + ...$

CT = "read bus to MBR"

- ICC - bits P and Q
- PQ = 00 Fetch Cycle
- PQ = 01 Indirect Cycle
- PQ = 10 Execute Cycle
- PQ = 11 Interrupt Cycle
Hardwired Control Unit

- Opcode decoder (4-to-16)
- 4-bit instruction code as input to CU
- Only one signal active at any given stage

Finite State Diagram
## State transitions

**Next state from current state**

<table>
<thead>
<tr>
<th>Current State</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>State 0</td>
<td>State 1</td>
</tr>
<tr>
<td>State 1</td>
<td>State 2, S6, S8, S10</td>
</tr>
<tr>
<td>State 2</td>
<td>S5 or ...</td>
</tr>
<tr>
<td>State 3</td>
<td>S9 or ...</td>
</tr>
<tr>
<td>State 4</td>
<td>State 0</td>
</tr>
<tr>
<td>State 5</td>
<td>State 0</td>
</tr>
<tr>
<td>State 6</td>
<td>State 7</td>
</tr>
<tr>
<td>State 7</td>
<td>State 0</td>
</tr>
<tr>
<td>State 8</td>
<td>State 0</td>
</tr>
<tr>
<td>State 9</td>
<td>State 11</td>
</tr>
<tr>
<td>State 10</td>
<td>State 11</td>
</tr>
<tr>
<td>State 11</td>
<td>State 0</td>
</tr>
</tbody>
</table>

**Alternatively, prior state & condition**

<table>
<thead>
<tr>
<th>Prior State</th>
<th>Condition</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S4, S5, S7, S8, S9, S11</td>
<td>State 0</td>
</tr>
<tr>
<td></td>
<td>State 0</td>
<td>State 1</td>
</tr>
<tr>
<td></td>
<td>State 0</td>
<td>State 2</td>
</tr>
<tr>
<td></td>
<td>State 0</td>
<td>State 3</td>
</tr>
<tr>
<td></td>
<td>State 0</td>
<td>State 4</td>
</tr>
<tr>
<td></td>
<td>State 5</td>
<td>State 6</td>
</tr>
<tr>
<td></td>
<td>State 6</td>
<td>State 7</td>
</tr>
<tr>
<td></td>
<td>State 8</td>
<td>State 9</td>
</tr>
<tr>
<td></td>
<td>State 9</td>
<td>State 10</td>
</tr>
<tr>
<td></td>
<td>State 10</td>
<td>State 11</td>
</tr>
</tbody>
</table>

## Hardwired Control Summary

- **Control signal generation in hardware is fast**
- **Weaknesses**
  - CU difficult to design
    - Circuit can become large and complex
  - CU difficult to modify and change
    - Design and ‘minimizing’ must be done again after every change
- **RISC-philosophy makes it a bit easier**
  - Simple instruction set makes the design and implementation easier
Microprogrammed Control
\textit{(Mikro-ohjelmoitu ohjaus)}

- Idea 1951: Wilkes Microprogrammed Control (Maurice Wilkes)
- Execution Engine
  - Execution of one machine instruction is done by executing a sequence of microinstructions (micro-operations)
  - Executes each microinstruction by generating the control signals indicated by the instruction
- Micro-operations stored in control memory as microinstructions
  - Firmware \textit{(laiteohjelmisto)}
- Each microinstruction has two parts
  - What is done during the coming clock cycle?
    - Microinstruction indicates the control signals
    - Deliver the control signals to circuits
  - What/where is the next microinstruction?
    - Assumption: next microinstruction from next location
    - Microinstruction can contain the address of next microinstruction!
Microinstructions

- Each stage in instruction execution cycle is represented by a sequence of microinstructions that are executed during the cycle in that stage.
- E.g. in ROM:
  - Microprogram or firmware

Microinstructions (Sta10 Fig 16.2)

Horizontal microinstruction

- All possible control signals are represented in a bit vector of each microinstruction:
  - One bit for each signal (1=generate, 0=do not generate)
  - Long instructions if plenty of signals used
- Each microinstruction is a conditional branch:
  - What status bit(s) checked
  - Address of the next microinstruction

Horizontal microinstruction (Sta10 Fig 16.1 a)
Vertical Microinstruction

- Control signals coded to number (function)
- Decode back to control signals during execution
- Shorter instructions, but decoding takes time
  - Gate delay?
- Each microinstruction is conditional branch (as with horizontal instructions)

Microinstruction Execution Engine

- Control Address Register, CAR
  - Which microinstruction next?
  - ~ instr. pointer, “μPC”
- Control memory
  - Microinstructions
    - fetch, indirect, execute, interrupt
- Control Buffer Register, CBR
  - Register for executing microinstr.
  - ~ instr. register, “μIR”
  - Generate the signals to circuits
    - Verticals through decoder
- Sequencing Logic
  - Next address to CAR
Which Microinstruction Next?

a) Explicit

- Each instruction has 2 addresses
  - With the conditions flags that are checked for branching
  - Next instruction from either address (select using the flags)
  - Often just the next location in control memory
    - Why store the address?
    - No time for addition!

b) Implicit

- Assumption: next microinstruction from next location in control memory
  - Must be calculated

- Instruction has 1 address
  - Still need condition flags
  - If condition=1, use the address

- Address part not always used
  - Wasted space
c) Variable format

- Some bits interpreted in two ways
  - 1 b: Address or not
  - Only branch instructions have address
  - Branch instructions do not have control signals
  - If jump, need to execute two microinstructions instead of just one
    - Wasted time?
    - Saved space?

Which Microinstruction Next?

- Wasted time?
- Saved space?

Which Microinstruction Next?

- How to locate the correct microinstruction routine?
  - Control signals depend on the current machine instruction
- Generate first microinstruction address from op-code (mapping + combining/adding)
  - Most-significant bits of address directly from op-code
  - Least-significant bits based on the current situation (0 or 1)
  - Example: IBM 3033 Control Address Register (CAR), 13 bit address
    - Op-code gives 8 bits -> each sequence 32 micro-instr.
    - rest 5 bits based on the certain status bits
Which Microinstruction Next?

e) Subroutines and residual control

- Microinstruction can set a special return register with 'return address'
  - No context, just one return allowed (one-level only)
  - No nested structure
  - Example: LSI-11, 22 bit microinstruction
    - Control memory 2048 instructions, 11 bit address
    - OP-code determines the first microinstruction address
    - Assumption, next is CAR ← CAR+1
    - Each instruction has a bit: subroutine call or not
    - Call:
      - Store return address (only the latest one available)
      - Jump to the routine (address in the instruction)
    - Return: jump to address in return register

Microinstruction Coding

- Horizontal or Vertical?
  - Horizontal: fast interpretation
  - Vertical: less bits, smaller space
- Often a compromise, using mixed model
  - Microinstruction split to fields, each field is used for certain control signals
  - Excluding signal combinations can be coded in the same field
  - Coding decoded to control signals during execution
    - One field can control decoding of other fields!
- Several shorter coded fields easier for implementation than one long field
  - Several simple decoders
Microinstruction Coding

- **Functional encoding** (toiminnotaitain)
  - Each field controls one specific action (e.g., load)
    - Load from accumulator
    - Load from memory
    - Load from ...

- **Resource encoding** (resursseittaitain)
  - Each field controls specific resource (e.g. accumulator)
    - Load from accumulator
    - Store to accumulator
    - Add to accumulator
    - … accumulator

Vertical vs. Horizontal Microcode

Next microinstruction address (CAR = CSAR)
Assumption: CAR = CAR + 1
Why microprogrammed control?

- … even when its slower than hardwired control
- Design is simple and flexible
  - Modifications (e.g. expansion of instruction set) can be added very late in the design phase
  - Old hardware can be updated by just changing control memory
    - Whole control unit chip in older machines (μcode chip)
  - There exists (existed?) development environments for microprograms
- Backward compatibility
  - Old instruction set can be used easily
  - Just add new microprograms for new machine instructions
- Generality
  - One hardware, several different instruction sets
  - One instruction set, several different organizations

Control Summary

- Control signals
- Hardwired control
- Microprogrammed control?
  - Control memory, control address, control buffer
  - Horizontal vs. vertical microprogrammed control?
  - How do you find the next microinstruction?
  - LSI-11 example
Review Questions / Kertauskysymyksiä

- Hardwired vs. microprogrammed control?
- How to determine the address of microinstruction?
- What is the purpose of control memory?
- Horizontal vs. vertical microinstruction?
- Compare microprogram execution to machine language fetch-execute cycle.
- Microprogrammed vs. hardwired?