Overview

• Control Unit:
  – CPU operations can be broken into smaller time scale "micro-operations"
  – Control unit co-ordinates these operations
  – Can be implemented as circuits, or micro-programmed.

• Pipeline:
  – Different instruction's Micro-operations can overlap each other
  – Achieve parallelism by having a "pipeline" of operations.

Control Unit, Pipeline

(Refer to chapter 16)

Processor Functional Spec

• Instruction set is still a high level definition, in more
detail, we need to know:
  – Operations (opcode)
  – Addressing modes
  – Registers
  – I/O interface
  – Memory module interface
  – Interrupt handling structure
Instruction Cycle Micro-Operations

- A computer executes a **program**
- A program has many **instruction cycles**
- Each instruction cycle has a number of steps (pipelines) called **micro-operations**

Fetch Cycle Micro-Ops

- Address of next instruction is in **PC**
- Address (MAR) is placed on address bus
- Control unit issues READ command
- Result (data from memory) appears on data bus
- Data from data bus copied into **MBR**
- **PC** incremented by 1 (in parallel with data fetch from memory)
- Data (instruction) moved from **MBR** to **IR**
- **MBR** is now free for further data fetches

Fetch Cycle - Registers involved

- **Memory Address Register (MAR)**
  - Connected to address bus
  - Specifies address for read or write op
- **Memory Buffer Register (MBR)**
  - Connected to data bus
  - Holds data to write or last data read
- **Program Counter (PC)**
  - Holds address of next instruction to be fetched
- **Instruction Register (IR)**
  - Holds last instruction fetched

Fetch Cycle Register States

1. **MAR**
   - 0000000001100100
2. **MBR**
   - 0001000001000000
3. **PC**
   - 0000000001100101
4. **IR**
   - 0000000001100100
5. **AC**
   - 0000000001100100

$t1 : \text{MAR} \leftarrow (PC)
$t2 : \text{MBR} \leftarrow \text{Memory}
$t3 : \text{PC} \leftarrow (PC) + 1
\text{IR} \leftarrow (\text{MBR})$
Rules for Clock Cycle Grouping

- **Proper sequence must be followed**
  - MAR <- (PC) must precede MBR <- (memory)

- **Conflicts must be avoided**
  - Must not read & write same register at same time
  - MBR <- (memory) & IR <- (MBR) must not be in same cycle

- **Also:** PC <- (PC) +1 involves addition
  - Use ALU
  - May need additional micro-operations

Indirect Cycle

- t1: Need to load data addr from memory
- t2: MBR contains an address
- t3: IR is now in same state as if direct addressing had been used

Interrupt Cycle

\[
\begin{align*}
  t1 & : \text{MBR} \leftarrow \text{PC} \\
  t2 & : \text{MAR} \leftarrow \text{SavePcToAddr} \\
  & \quad \text{PC} \leftarrow \text{IsrAddr} \\
  t3 & : \text{Memory} \leftarrow (\text{MBR})
\end{align*}
\]

- **This is a minimum**
  - Steps t1, t2: save current PC addr
  - Step 3: store MBR, which is the old value of PC

Execute Cycle

- Fetch and Interrupt Cycles's micro-ops are fixed
- Micro-ops for execution cycle is different for each instruction
Execute Cycle (ADD)

- ADD R1,X - add the contents of location X to Register 1, result in R1
- t1: MAR ← IR(X)
- t2: MBR ← (memory)
- t3: R1 ← R1 + (MBR)
- Note no overlap of micro-operations

Execute Cycle (ISZ)

- ISZ X - increment and skip if zero
  - t1: MAR ← (IR(X))
  - t2: MBR ← (memory)
  - t3: MBR ← (MBR) + 1
  - t4: memory ← (MBR)
  - if (MBR) == 0 then PC ← (PC) + 1
- Notes:
  - if is a single micro-operation
  - Micro-operations done during t4

Execute Cycle (BSA)

- BSA X - Branch and save address
  - Address of instruction following BSA is saved in X
  - Execution continues from X+1
  - t1: MAR ← (IR(X))
  - MBR ← (PC)
  - t2: PC ← (IR(X))
  - memory ← (MBR)
  - t3: PC ← (PC) + 1

Instruction Cycle

- Each phase decomposed into sequence of elementary micro-operations
- E.g. fetch, indirect, and interrupt cycles
- Execute cycle
  - One sequence of micro-operations for each opcode
  - Can be different with different number of micro-ops
- Need to tie sequences together
- Assume a new 2-bit register
  - Instruction cycle code (ICC) designates which part of cycle processor is in
    » 00: Fetch
    » 01: Indirect
    » 10: Execute
    » 11: Interrupt
**Flowchart for Instruction Cycle**

**Processor Control**

- How to implement controls of CPU?
  - What are the basic elements in CPU?
  - Micro-operations involved?
  - Functions and logic/circuits that implement these micro-operations and sequences

**Basic Elements of Processor**

- ALU
  - Operates on registers with Arithmetic & Logic functions

- Registers
  - Internal CPU storage

- Internal data paths
  - Connects registers, ALU

- External data paths
  - Connects registers and memory

- Control Unit

**Types of Micro-operation**

- Transfer data between registers
  - e.g., move $R1, $R2

- Transfer data from register to external
  - E.g., sw $R1, $ra

- Transfer data from external to register
  - lw r1, $r0

- Perform arithmetic or logical ops
  - Operates on registers
Functions of Control Unit

• **Sequencing**
  – Causing the CPU to step through a series of micro-operations

• **Execution**
  – Causing the performance of each micro-op

• **This is done using Control Signals**

### Control Signal Input

- **Clock**
  – One micro-instruction (or set of parallel micro-instructions) per clock cycle

- **Instruction register**
  – Op-code for current instruction
  – Determines which micro-instructions are performed

- **Flags**
  – State of CPU
  – Results of previous operations

- **From control bus**
  – Interrupts
  – Acknowledgements

### Control Signals Output

- **Within CPU**
  – Cause data movement among registers
  – Activate specific functions, eg, ALU

- **Via control bus**
  – To memory
  – To I/O modules
Model of Control Unit

Example Control Signal Sequence - Fetch

- **MAR <- (PC)**
  - Control unit activates signal to open gates between PC and MAR

- **MBR <- (memory)**
  - Open gates between MAR and address bus
  - Memory read control signal
  - Open gates between data bus and MBR

Control Signal Circuits Example

- **Micro-Ops and Control signals Fetch, Indirect and Int**

<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>t_1: MAR &lt;- (PC)</td>
<td></td>
<td>C_2</td>
</tr>
<tr>
<td>t_2: MBR &lt;- Memory</td>
<td></td>
<td>C_9, C_R</td>
</tr>
<tr>
<td></td>
<td>PC &lt;- (PC) + 1</td>
<td></td>
</tr>
<tr>
<td>t_3: IR &lt;- (MBR)</td>
<td></td>
<td>C_4</td>
</tr>
<tr>
<td>Indirect:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t_1: MAR &lt;- (IR(Address))</td>
<td></td>
<td>C_9</td>
</tr>
<tr>
<td>t_2: MBR &lt;- Memory</td>
<td></td>
<td>C_9, C_R</td>
</tr>
<tr>
<td>t_3: IR(Address) &lt;- (MBR(Address))</td>
<td></td>
<td>C_4</td>
</tr>
<tr>
<td>Interrupt:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t_1: MBR &lt;- (PC)</td>
<td></td>
<td>C_1</td>
</tr>
<tr>
<td>t_2: MAR &lt;- Save-address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC &lt;- Routine-address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t_3: Memory &lt;- (MBR)</td>
<td></td>
<td>C_1, C_W</td>
</tr>
</tbody>
</table>
Internal Bus Organization

- Usually a single internal bus
- Gates control movement of data onto and off the bus
- Control signals control data transfer to and from external systems bus
- Temporary registers needed for proper operation of ALU

Control Unit Implementation

- Clocks, Flags, control bus signals:
  - Each bit is clearly defined, can be implemented by logics
- Opcode from IR:
  - Need to generate a sequence of control signals depending on opcode
  - Implemented by a decoder on Opcode

Hardwired Implementation (1)

- Control unit inputs
- Flags and control bus
  - Each bit means something
- Instruction register
  - Op-code causes different control signals for each different instruction
  - Unique logic for each op-code
  - Decoder takes encoded input and produces single output
  - \(n\) binary inputs and \(2^n\) outputs

Hardwired Implementation (2)

- Clock
  - Repetitive sequence of pulses
  - Useful for measuring duration of micro-ops
  - Must be long enough to allow signal propagation
  - Different control signals at different times within instruction cycle
  - Need a counter with different control signals for t1, t2 etc.
Control Unit Implementation

- Clocks, Flags, control bus signals:

![Control Unit Example](image)

Intel 8085 CPU Block Diagram

Intel 8085 control unit
Problems With Hard Wired Designs

- Complex sequencing & micro-operation logic
- Difficult to design and test
- Inflexible design
- Difficult to add new instructions
- Solution: Micro-program

Micro-Programmed Control

- A brief intro:
  - No hardwiring of control signals to the input Flag/Clock/Instruction
  - Load control signals from control memories
  - Controlled by control addr reg

Control Unit Summary

- Control unit handles CPU operations and communications with Registers and Memory
- Input: Instructions, Clock, and Flags
- Output: control signals and sequences that allow instruction to be fetched, operands loaded and instruction executed correctly
- Can be implemented as hardwired circuits
- Can also be implemented as micro-programmed controls.
Operation Cycles of Instructions

- A typical instruction cycle will typically have:
  - Fetch Instruction (FI): load instruction
  - Decode Instruction (DI): understand what to do
  - Calculate Operands (CO): calculate the effective address of operands
  - Fetch Operands (FO): fetch operands from memory
  - Execute Instruction (EI): generate necessary control signals and sequences to finish operation on operands in registers
  - Write Operands (WO): write operand from register to memory

Data Flows

- Fetch Cycle Data Flow:
Data Flows

**Indirect Cycles:**
- Load operands that are not in registers
- Addr of operand in MBR $\Rightarrow$ MAR

---

Data Flows

**Interrupt Handling Cycles:**
- Save current PC to MBR, and then to mem (e.g., pointed by SP)

---

How to speed up instruction execution?

**Question:**
- Do we have to wait for all cycles to complete before starting the next instruction?

**Answer:**
- No, we can do better
- It is called **Pipeline**
Instruction Micro-Operations

- **An 6-stage pipeline**
  - Execution takes longer than fetch
  - Break up execution into sub-cycles, i.e, DI, CO, FO, EI, WO.
  - Allow overlapping, or pre-fetch the command
  - Branch: may have to re-fetch the correct instruction

<table>
<thead>
<tr>
<th>Instruction 1</th>
<th>Instruction 2</th>
<th>Instruction 3</th>
<th>Instruction 4</th>
<th>Instruction 5</th>
<th>Instruction 6</th>
<th>Instruction 7</th>
<th>Instruction 8</th>
<th>Instruction 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>FI</td>
<td>DI</td>
<td>CO</td>
<td>FO</td>
<td>EI</td>
<td>WO</td>
<td>FI</td>
<td>DI</td>
<td>CO</td>
</tr>
</tbody>
</table>

Speedup: 9x6=54 (no pipeline) vs 14 (pipelined) time slots.

Pipeline efficiency

- **Non-uniform pipeline stages**
  - Involves waiting at different stages
- **Conditional branch**
  - Pre-fetched instructions become invalid
- **Data dependency**
  - Instruction k writes to memory, instruction k+1 reads the data
  - Then need to wait instruction k’s WO to finish,
- **The more stages the better?**
  - No, additional data moving slows down the execution, plus circuits complexity

Pipeline Hazards

- **Structural**:
  Structural hazzards occur when the same functional unit is needed to be used by two different instructions at the same time in the same cycle.
- **Control**:
  Branches and jumps (subroutine calls) disrupt the sequence of instructions being issued and may result in a pipeline stall. Early branch scheduling and branch prediction are used to minimize the number of stalls.
- **Data**:
  Data hazzards occur when an instruction reads from a register before an earlier instruction has written the expected value into the register (WAR),
Conditional branching

• The correct PC address is runtime dependent

| Instruction 1 | FI | DI | CO | FO | EI | WO |
| Instruction 2 | FI | DI | CO | FO | EI | WO |
| Instruction 3 | FI | DI | CO | FO | EI | WO |
| Instruction 4 | FI | DI | CO | FO | EI | WO |
| Instruction 5 | FI | DI | CO | FO | EI | WO |
| Instruction 6 | FI | DI | CO | FO | EI | WO |
| Instruction 7 | FI | DI | CO | FO | EI | WO |
| Instruction 15 | FI | DI | CO | FO | EI | WO |
| Instruction 16 | FI | DI | CO | FO | EI | WO |

Branch

Alternative Pipeline View

Found that Correct PC should be I15
Flush out I6-I3

Pipeline Efficiency Analysis

• K-stage pipeline, n instructions, execution time:

\[ T_{k,n} = \left[ k + (n-1) \right] \tau \]

• Speed up factor as function of stages

\[ S_k = \frac{T_{1,n}}{T_{k,n}} = \frac{nk \tau}{\left[ k + (n-1) \right] \tau} = \frac{nk}{k + (n-1)} \]
Dealing with Branches

• Pipeline efficiency depends on a steady stream of instructions that fills up the pipeline
• Conditional branching is a major drawback for efficiency
• Can be dealt with by:
  – Multiple Streams
  – Prefetch Branch Target
  – Loop buffer
  – Branch prediction
  – Delayed branching

Multiple Streams

• Have two pipelines
• Prefetch each branch into a separate pipeline
• Use appropriate pipeline
• Leads to bus & register contention
• Multiple branches lead to further pipelines being needed

Prefetch Branch Target

• Target of branch is prefetched in addition to the instructions following branch (PC+1)
• Keep target until branch is executed
• Used by IBM 360/91

Loop Buffer

• Very fast memory in CPU
• Maintained n most recently fetched instructions
• Check buffer before fetching from memory
• Very good for small loops or jumps
  – E.g. typical IF THEN, IF THEN ELSE sequences
  – Like cache for instructions.
  – Used in supercomputer like CRAY-1
Branch Prediction - Static Solutions

• Predict never taken
  – Assume that jump will not happen
  – Always fetch next instruction
  – 68020 & VAX 11/780

• Predict always taken
  – Assume that jump will happen
  – Always fetch target instruction

• Predict by opcode
  – By collecting stats on different opcode w.r.t. branching
  – Correct rate > 75%

Branch Prediction - Dynamic, Runtime Based

• Taken/Not taken switch
  – Use 1 or 2 bits to record taken/not taken history
  – Good for loops

• Branch history table
  – Based on previous history
  – Good for loops

Branch Prediction State Diagram

Examples of Pipeline designs
MIPS-Microprocessor w/o Interlocked Pipeline Stages

- IF—First half of instruction fetch, PC selection actually happens here, together with initiation of instruction cache access.
- IS—Second half of instruction fetch, complete instruction cache access.
- RF—Instruction decode and register fetch, hazard checking, and also instruction cache hit detection.
- EX—Execution, which includes effective address calculation, ALU operation, and branch-target computation and condition evaluation.
- DF—Data fetch, first half of data cache access.
- DS—Second half of data fetch, completion of data cache access.
- TC—Tag check, determine whether the data cache access hit.
- WB—Write back for loads and register-register operations.

Delay from load stalls

- 2 cycle delay for load data:
  - Data available only at the end of DS (cc 6 for instruction 1)

Floating point op cycles

- 3-cycle delay for branch
  - Target addr available after EX
Pipeline Summary

- **Pipeline is ILP - Instruction Level Parallelism**
  - Could potentially achieve \( k \) time speed up for \( k \)-stage pipelines

- **Pipeline Hazards:**
  - Structural: two micro-ops requires the same circuits in the same cycle
  - Control: target branch PC not known until execution
  - Data: successive instructions read the output of previous instruction