Processor Data Paths - ALU and Registers

Incorporating the ALU into a Processor Data Path
L4 – Incorporating ALU into data path

- Building up the data path
- Control of the data path
Have our multifunction ALU

- Will now put an interface around the ALU and build up a data path
The wrapper

- Establish an interface to the ALU slices
- Have the data inputs
- Have a Carry in
- Have control in
- Produce
  - The result R
  - A carry out
Build up step 1

- Add input and output latches/bus drivers
- Add 2 internal data busses
- Add flag generation
- Inputs & control – A, B, Cin, Pctl, Kctl, Rctl
- Outputs – Cout, R, Nflag, Zflag
Build up step 2

- Now add a general purpose registers
- These are dual ported registers
- Loading or driving of registers on the bus is controlled by the clock
- Busses are tristate
Dual Ported Registers

- Register are dual ported
  - Registers can be loaded from or drive a value onto both the A Bus and B Bus simultaneously
  - Possible combinations on same cycle
    - Drive both the A Bus and B Bus
    - Load from both the A Bus and B Bus
      - Must be different registers!!!!!!!
      - Register bank does not check for this – responsibility of the controller
    - Load from one Bus and drive the other
Now add the controller

- Note the Instruction Register and Flags register
Program Counter and MDR, MAR

- MDR – Memory Data Register
- MAR – Memory Address Register
- PC Register increment and offset integrated into register design
Other Functional Units

- May want to incorporate
  - Shifter
  - Floating Point Units
  - Index addressing registers
  - Etc.
General Operation of a Datapath

- Datapaths are generally synchronous
- Internal clock most likely higher than bus clock
- Typical RTL of datapath
  - Cyc 1: Mem(PC) -> IR
  - PC + 1 -> PC
  - Cyc 2: RS1+RS2 -> ALUout
  - Cyc 3: ALU out -> RS1
Datapath Control Evolution

- Finite State Machine (FSM)
  - Fixed and unvarying action of a datapath
  - FSM implemented with PLA

![Diagram of Finite State Machine (FSM)](image-url)

Fig. 6.1 Finite-state machine controlling the data path. In this case there is periodic cycling through a fixed sequence of states.
Use of flags

- Here the current data can affect the action of the controller

Fig. 6.2 Finite-state machine controlling the data path. In this case the next state can be a function of the previous operation’s outcome.
Load flags

- By loading a flags register can use them to affect the controller only at specific points in the “program”
Data from memory used

- In addition to flags, data from memory can affect the state and the datapaths action
Instruction Reg

- Have now progressed to a “stored” program
Alternative form

- Here use a ROM rather than a PLA finite state machine

Fig. 6.7 An alternative form of stored-program machine, illustrating the use of a decoder and memory to implement the state machine controlling the data path.
Another alternative

- Use a microprogram counter
A different slice

- Just a different visualization

Fig. 6.9 Another way of visualizing the machine shown in Fig. 6.8.
Operation

- Shows steps the controller would go through to execute the instructions

Fig. 6.6 A portion of the state diagram for the controller of a stored-program machine, illustrating some typical instruction interpretation state sequences and their associated control outputs.
# Operation Examples – ALU op

<table>
<thead>
<tr>
<th>Function of sub-sequence</th>
<th>Control [&amp; State] sequence</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch Next Inst:</td>
<td>RPORT ← PC</td>
<td>Place next instr. address in right port.</td>
</tr>
<tr>
<td></td>
<td>read memory</td>
<td>Raise control line to initiate memory read.</td>
</tr>
<tr>
<td></td>
<td>PC ← PC + 1</td>
<td>Increment PC, overlapping incr. with fetch.</td>
</tr>
<tr>
<td></td>
<td>[Y’ = fcn(memop complete)]</td>
<td>Loop here till memory read completes.</td>
</tr>
<tr>
<td></td>
<td>IR ← mem data</td>
<td>Load IR with inst, when read completed.</td>
</tr>
<tr>
<td>Decode Instruction:</td>
<td>[Y’ = fcn(IR)]</td>
<td>Set machine state as fcn of instruction.</td>
</tr>
<tr>
<td>Fetch Operands:</td>
<td>A ← R7</td>
<td>Load ALU input registers with operands.</td>
</tr>
<tr>
<td></td>
<td>B ← R2</td>
<td></td>
</tr>
<tr>
<td>Store Result:</td>
<td>R5 ← ALUoutreg</td>
<td>Send result address to R5.</td>
</tr>
<tr>
<td></td>
<td>[Y’ = FNI]</td>
<td>Inst. not a branch, so simply return to FNI state.</td>
</tr>
</tbody>
</table>
Store result back to memory

<table>
<thead>
<tr>
<th>Function of sub-sequence</th>
<th>Control [&amp; State] sequence</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch Next Inst:</td>
<td>RPORT ← PC</td>
<td>Place next instr. address in right port.</td>
</tr>
<tr>
<td></td>
<td>read memory</td>
<td>Raise control line to initiate memory read.</td>
</tr>
<tr>
<td></td>
<td>PC ← PC + 1</td>
<td>Increment PC.</td>
</tr>
<tr>
<td></td>
<td>[Y’ = fcn(memop complete)]</td>
<td>Loop here till memory read completes.</td>
</tr>
<tr>
<td></td>
<td>IR ← mem data</td>
<td>Load IR with inst, when read completed.</td>
</tr>
<tr>
<td>Decode Instruction:</td>
<td>[Y’ = fcn(IR)]</td>
<td>Set machine state as fcn of instruction.</td>
</tr>
<tr>
<td>Perform Operation:</td>
<td>RPORT ← IR(ADDRESS)</td>
<td>Send the contents of IR address field to memory.</td>
</tr>
<tr>
<td></td>
<td>write memory</td>
<td>Raise write control line to init. memory write.</td>
</tr>
<tr>
<td>Store Result:</td>
<td>RPORT ← R3</td>
<td>Place data in right output port.</td>
</tr>
<tr>
<td></td>
<td>[Y’ = fcn(memop complete)]</td>
<td>Loop here till memory write completes.</td>
</tr>
<tr>
<td></td>
<td>[Y’ = FNI]</td>
<td>Inst. not a branch, so simply return to FNI state.</td>
</tr>
</tbody>
</table>