Approaching an ISA

- Instruction Set Architecture
  - Defines set of operations, instruction format, hardware supported data types, named storage, addressing modes, sequencing
  - Meaning of each instruction is described by RTL on architected registers and memory
  - Given technology constraints assemble adequate datapath
    - Architected storage mapped to actual storage
    - Function units to do all the required operations
    - Possible additional storage (e.g., MAR, MBR, ...)
    - Interconnect to move information among regs and FUs
  - Map each instruction to sequence of RTLs
  - Collate sequences into symbolic controller STD
  - Lower symbolic STD to control points
  - Implement controller

Outline

- Review: high level optimization of the list processor
- General notion of instruction execution cycle and the pieces that perform it
- ISA => implementation
- Example
- Generalize and discuss

Resource Utilization Charts

- One way to visualize datapath optimizations is through the use of a resource utilization charts.
- These are used in high-level design to help schedule operations on shared resources.
- Resources are listed on the y-axis. Time (in cycles) on the x-axis.
- Example:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>A1+B1</td>
<td>A2+B2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cycle</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

- Our list processor has two shared resources: memory and adder

List Example Resource Scheduling

- **Unoptimized solution:**
  1. \( \text{SUM} + \text{Memory}(\text{NEXT+1}) \)
  2. \( \text{NEXT} + \text{Memory}(\text{NEXT}) \)

<table>
<thead>
<tr>
<th>Resource</th>
<th>Fetch A</th>
<th>Fetch B</th>
<th>Add A</th>
<th>Add B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>Next</td>
<td>Next</td>
<td>Sum</td>
<td>Sum</td>
</tr>
<tr>
<td>Add</td>
<td>Sum</td>
<td>Sum</td>
<td>Sum</td>
<td>Sum</td>
</tr>
</tbody>
</table>

- **Optimized solution:**
  1. \( \text{SUM} + \text{Memory}(\text{NUMA}) \)
  2. \( \text{NEXT} + \text{Memory}(\text{NEXT}) \)

<table>
<thead>
<tr>
<th>Resource</th>
<th>Fetch A</th>
<th>Fetch B</th>
<th>Add A</th>
<th>Add B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>Next</td>
<td>Next</td>
<td>Sum</td>
<td>Sum</td>
</tr>
<tr>
<td>Add</td>
<td>Sum</td>
<td>Sum</td>
<td>Sum</td>
<td>Sum</td>
</tr>
</tbody>
</table>

- **How about the other combination:**
  \( \text{add} \times \text{register} \)

<table>
<thead>
<tr>
<th>Resource</th>
<th>Fetch A</th>
<th>Fetch B</th>
<th>Add A</th>
<th>Add B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>Next</td>
<td>Next</td>
<td>Num</td>
<td>Num</td>
</tr>
<tr>
<td>Add</td>
<td>Num</td>
<td>Num</td>
<td>Sum</td>
<td>Sum</td>
</tr>
</tbody>
</table>

- **Does this work?** If so, a very short clock period. Each cycle could have independent fetch and add. \( T = \max( T_{\text{mem}}, T_{\text{add}} ) \) Instead of \( T_{\text{mem}} + T_{\text{add}} \)
**Instruction Sequencing**

- Example – an instruction to add the contents of two registers (Rx and Ry) and place result in a third register (Rz)
- Step 1: Fetch the ADD instruction from memory into an instruction register
- Step 2: Decode instruction
  - Instruction in IR has the code of an ADD instruction
  - Register indices used to generate output enable for registers Rx and Ry
  - Register index used to generate load signal for register Rz
- Step 3: Execute instruction
  - Enable Rx and Ry output and direct to ALU
  - Setup ALU to perform ADD operation
  - Direct result to Rz so that it can be loaded into register

**Instruction Types**

- Data Manipulation
  - Add, subtract
  - Increment, decrement
  - Multiply
  - Shift, rotate
  - Immediate operands
- Data Staging
  - Load/store data to/from memory
  - Register-to-register move
- Control
  - Conditional/unconditional branches in program flow
  - Subroutine call and return

**Elements of the Control Unit (aka Instruction Unit)**

- Standard FSM Elements
  - State register
  - Next-state logic
  - Output logic (datapath/control signaling)
  - Moore or synchronous Mealy machine to avoid loops unbroken by FF
- Plus Additional "Control" Registers (in DP)
  - Instruction register (IR)
  - Program counter (PC)
- Inputs/Outputs
  - Outputs control elements of data path
  - Inputs from data path used to alter flow of program (test if zero)

**Instruction Execution**

- Control State Diagram (for each diagram)
  - Reset
  - Fetch instruction
  - Decode
  - Execute
- Instructions partitioned into three classes
  - Branch
  - Load/store
  - Register-to-register
- Different sequence through diagram for each instruction type
  - Controller manipulates the data path to perform the instruction

**Data Path (Hierarchy)**

- Arithmetic circuits constructed in hierarchical and iterative fashion
  - each bit in datapath is functionally identical
  - 4-bit, 8-bit, 16-bit, 32-bit datapaths

**Data Path (ALU)**

- ALU Block Diagram
  - Input: data and operation to perform
  - Output: result of operation and status information
**Data Path (ALU + Registers + interconnect)**

- **Accumulator**
  - Special register
  - One of the inputs to ALU
  - Output of ALU stored back in accumulator

- **One-address instructions**
  - Operation and address of one operand
  - Other operand and destination is accumulator register
  - AC ← AC op Mem( addr )
  - "Single address instructions" (AC implicit operand)

- **Multiple registers**
  - Part of instruction used to choose register operands

**Instruction Path**

- **Program Counter**
  - Keeps track of program execution
  - Address of next instruction to read from memory
  - May have auto-increment feature or use ALU

- **Instruction Register**
  - Current instruction
  - Includes ALU operation and address of operand
  - Also holds target of jump instruction
  - Immediate operands

- **Relationship to Data Path**
  - Contents of IR may also be required as input to ALU
  - Literals, address offsets
  - Contents of PC used in branch target calculation

- **Relationship to controller**
  - Causes IR ← mem[PC]
  - IR contains OPCODE, which dictate controller outputs

**Data Path (Bit-slice)**

- **Bit-slice concept: iterate to build n-bit wide datapaths**

**Data Path (Memory Interface)**

- **Memory**
  - Separate data and instruction memory (Harvard architecture)
    - Two address busses, two data busses
  - Single combined memory (Princeton architecture)
  - Single address bus, single data bus

- **Separate memory**
  - ALU output goes to data memory input
  - Register input from data memory output
  - Data memory address from instruction register
  - Instruction register from instruction memory output
  - Instruction memory address from program counter

- **Single memory**
  - Address from PC or IR
  - Memory output to instruction and data registers
  - Memory input from ALU output

**Block Diagram of Processor**

- **Register Transfer View of Princeton Architecture**
  - Which register outputs are connected to which register inputs
  - Arrows represent data-flow, other are control signals from control FSM
  - MAR may be a simple multiplexer rather than separate register
  - MBR is split in two (REG and IR)
  - Load control for each register

- **Register Transfer View of Harvard Architecture**
  - Which register outputs are connected to which register inputs
  - Arrows represent data-flow, other are control signals from control FSM
  - Two MARs (PC and IR)
  - Two MBRs (REG and IR)
  - Load control for each register
A simplified Processor Data-path and Memory

- Princeton architecture
- Register file
- Instruction register
- PC incremented through ALU
- Modeled after MIPS r000 (used in 61C textbook by Patterson & Hennessy)
  - Really a 32 bit machine
  - We’ll do a 16 bit version

memory has only 255 words with a display on the last one

Tracing an Instruction’s Execution

- Instruction: `r3 = r1 + r2`
  - R 0 rs=1 rt=2 rd=3 funct=0

  1. Instruction fetch
     - Move instruction address from PC to memory address bus
     - Assert memory read
     - Move data from memory data bus into IR
     - Configure ALU to add to PC
     - Configure PC to store new value from ALUOut

  2. Instruction decode
     - Op-code bits of IR are input to control FSM
     - Rest of IR bits encode the operand addresses (rs and rt)
       - These go to register file

  3. Instruction execute
     - Set up ALU inputs
     - Configure ALU to perform ADD operation
     - Configure register file to store ALU result (rd)

Processor Control

- Synchronous Mealy machine
- Multiple cycles per instruction

Example: Processor Instructions

- Three principal types (16 bits in each instruction)

<table>
<thead>
<tr>
<th>type</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Register)</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>(Immediate)</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>7</td>
<td>Jump</td>
</tr>
</tbody>
</table>

- Some of the instructions

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>rs</td>
<td>rt</td>
</tr>
<tr>
<td>add</td>
<td>rs</td>
<td>rt</td>
</tr>
<tr>
<td>bc</td>
<td>rs</td>
<td>rt</td>
</tr>
<tr>
<td>addi</td>
<td>rs</td>
<td>rt</td>
</tr>
<tr>
<td>halt</td>
<td>7</td>
<td>-</td>
</tr>
</tbody>
</table>

Announcements

- Reading: 11.3 and 12.1
- HW 9 due Monday 2:10 pm
  - Check updated handout
- Digital Design in the News:
  - NY Times, NPR etc. 11-15
  - J. Stephen Smith - fluidic self-assembly for low-cost RFID tags
  - Another side of Moore’s Law
  - Power, power, power...
Tracing an Instruction's Execution (cont'd)

• Step 1

• Step 2

• Step 3

Register-Transfer-Level Description (cont'd)

• Control
- Transfer data between registers by asserting appropriate control signals
- Register transfer notation: work from register to register
  - Instruction fetch:
    - memory read: assert memory read signal (mr, RegBmEN)
    - IR read: load IR from memory data bus (IRld)
  - op = add: send PC into A input, 1 into B input, add (srcA, srcB0, srcB1, op)
    - PC = ALUout: load result of incrementing in ALU into PC (PCld, PCsel)
    - Instruction decode:
      - values of A and B read from register file (rs, rt)
    - Instruction execution:
      - op = add: send regA into A input, regB into B input, add (srcA, srcB0, srcB1, op)
      - rd = ALUout: store result of add into destination register (regWrite, wrDataSel, wrRegSel)

• How many states are needed to accomplish these transfers?
  - Data dependencies (where do values that are needed come from?)
  - Resource conflicts (ALU, busses, etc.)

• In our case, it takes three cycles
  - One for each step
  - All operation within a cycle occur between rising edges of the clock

• How do we set all of the control signals to be output by the state machine?
  - Depends on the type of machine (Mealy, Moore, synchronous Mealy)

Review of FSM Timing

to configure the data-path to do this here, when do we set the control signals?
FSM Controller for CPU (skeletal Moore FSM)

- First pass at deriving the state diagram (Moore machine)
  - These will be further refined into sub-states

![Diagram of FSM Controller for CPU](image)

FSM Controller for CPU (reset and instruction fetch)

- Assume Moore machine
  - Outputs associated with states rather than arcs
- Reset state and instruction fetch sequence
  - On reset (go to Fetch state)
    - Start fetching instructions
    - PC will set itself to zero
      - makes $\text{PC}$; memory read;
    - IR = memory data bus;
    - $\text{PC} \leftarrow \text{PC} + 1$;

![Diagram of FSM Controller for CPU (reset and instruction fetch)](image)

FSM Controller for CPU (decode)

- Operation Decode State
  - Next state branch based on operation code in instruction
  - Read two operands out of register file
    - What if the instruction doesn’t have two operands?

![Diagram of FSM Controller for CPU (decode)](image)

FSM Controller for CPU (Instruction Execution)

- For add instruction
  - Configure ALU and store result in register
    - $\text{rd} \leftarrow A + B$
  - Other instructions may require multiple cycles

![Diagram of FSM Controller for CPU (Instruction Execution)](image)

FSM Controller for CPU (Add Instruction)

- Putting it all together and closing the loop
  - the famous
  - Instruction fetch
    - decode
    - execute cycle

![Diagram of FSM Controller for CPU (Add Instruction)](image)

FSM Controller for CPU

- Now we need to repeat this for all the instructions of our processor
  - Fetch and decode states stay the same
  - Different execution states for each instruction
    - Some may require multiple states if available register transfer paths require sequencing of steps
**Approach an ISA**

- Instruction Set Architecture
  - Defines set of operations, instruction format, hardware supported data types, named storage, addressing modes, sequencing
- Meaning of each instruction is described by RTL on architected registers and memory
- Given technology constraints assemble adequate datapath
  - Architected storage mapped to actual storage
  - Function units to do all the required operations
  - Possible additional storage (eg. MAR, MBR, ...)
  - Interconnect to move information among regs and FUs
- Map each instruction to sequence of RTls
- Collate sequences into symbolic controller STD
- Lower symbolic STD to control points
- Implement controller

**Discussion**

- How would enhancing the datapath simplify control
  - Instruction and data access
  - PC arithmetic separate from ALU
  - Register file ports
- What determines the cycle time