Is organic computing finally here?

SPOTLIGHT | NOVEMBER 2, 2011

Today, computer chips can process data at 10 trillion (10\(^{13}\)) bits per second. But, even though neurons in the human brain fire at a rate of 100 times per second, the brain still outperforms the best computers at various tasks. The main reason being that calculations done by computer chips happen in isolated pipelines one at a time.

Japanese scientists have made organic molecules perform parallel computations like neurons in the human brain. They created this promising new approach with a ring-like molecule called 2,3-dichloro-5,6-dicyano-p-benzoquinone, or DDQ.

CS 61C: Great Ideas in Computer Architecture (Machine Structures)
Lecture 30: Pipeline Parallelism 1

Instructors:
Mike Franklin
Dan Garcia

http://inst.eecs.Berkeley.edu/~cs61c/fa11
Datapath Control Signals

- **ExtOp**: “zero”, “sign”
- **ALUsrc**: 0 ⇒ regB; 1 ⇒ immed
- **ALUctr**: “ADD”, “SUB”, “OR”

- **MemWr**: 1 ⇒ write memory
- **MemtoReg**: 0 ⇒ ALU; 1 ⇒ Mem
- **nPC_sel**: 0 ⇒ “+4”; 1 ⇒ “br”
- **RegDst**: 0 ⇒ “rt”; 1 ⇒ “rd”
- **RegWr**: 1 ⇒ write register

Where Do Control Signals Come From?

Instruction<31:0>
Summary of the Control Signals (1/2)

<table>
<thead>
<tr>
<th>Inst</th>
<th>Register Transfer</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>R[rd] ← R[rs] + R[rt]; PC ← PC + 4</td>
</tr>
<tr>
<td></td>
<td>ALUsrc=RegB, ALUctr=&quot;ADD&quot;, RegDst=rd, RegWr, nPC_sel=&quot;+4&quot;</td>
</tr>
<tr>
<td>sub</td>
<td>R[rd] ← R[rs] – R[rt]; PC ← PC + 4</td>
</tr>
<tr>
<td></td>
<td>ALUsrc=RegB, ALUctr=&quot;SUB&quot;, RegDst=rd, RegWr, nPC_sel=&quot;+4&quot;</td>
</tr>
<tr>
<td>ori</td>
<td>R[rt] ← R[rs] + zero_ext(Imm16); PC ← PC + 4</td>
</tr>
<tr>
<td></td>
<td>ALUsrc=Im, Extop=&quot;z&quot;, ALUctr=&quot;OR&quot;, RegDst=rt, RegWr, nPC_sel=&quot;+4&quot;</td>
</tr>
<tr>
<td>lw</td>
<td>R[rt] ← MEM[ R[rs] + sign_ext(Imm16)]; PC ← PC + 4</td>
</tr>
<tr>
<td></td>
<td>ALUsrc=Im, Extop=&quot;sn&quot;, ALUctr=&quot;ADD&quot;, MemToReg, RegDst=rt, RegWr, nPC_sel = &quot;+4&quot;</td>
</tr>
<tr>
<td>sw</td>
<td>MEM[ R[rs] + sign_ext(Imm16)] ← R[rs]; PC ← PC + 4</td>
</tr>
<tr>
<td></td>
<td>ALUsrc=Im, Extop=&quot;sn&quot;, ALUctr = &quot;ADD&quot;, MemWr, nPC_sel = &quot;+4&quot;</td>
</tr>
<tr>
<td>beq</td>
<td>if (R[rs] == R[rt]) then PC ← PC + sign_ext(Imm16)</td>
</tr>
<tr>
<td></td>
<td>else PC ← PC + 4</td>
</tr>
<tr>
<td></td>
<td>nPC_sel = &quot;br&quot;, ALUctr = &quot;SUB&quot;</td>
</tr>
</tbody>
</table>
Summary of the Control Signals (2/2)

See Appendix A

<table>
<thead>
<tr>
<th></th>
<th>func&lt;31:0&gt;</th>
<th>10 0000</th>
<th>10 0010</th>
<th>We Don't Care :)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00 0000</td>
<td>00 0000</td>
<td>00 1101</td>
<td>10 0011</td>
</tr>
<tr>
<td>RegDst</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ALUSrc</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MemtoReg</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>RegWrite</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MemWrite</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>nPCsel</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Jump</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ExtOp</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ALUctr&lt;2:0&gt;</td>
<td>Add</td>
<td>Subtract</td>
<td>Or</td>
<td>Add</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R-type</th>
<th>op&lt;5:0&gt;</th>
<th>rs&lt;5:0&gt;</th>
<th>rt&lt;5:0&gt;</th>
<th>rd&lt;5:0&gt;</th>
<th>shamt&lt;5:0&gt;</th>
<th>funct&lt;5:0&gt;</th>
<th>add, sub</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-type</td>
<td>op&lt;5:0&gt;</td>
<td>rs&lt;5:0&gt;</td>
<td>rt&lt;5:0&gt;</td>
<td></td>
<td>immediate&lt;5:0&gt;</td>
<td></td>
<td>ori, lw, sw, beq</td>
</tr>
<tr>
<td>J-type</td>
<td>op&lt;5:0&gt;</td>
<td></td>
<td></td>
<td>jump&lt;5:0&gt;</td>
<td></td>
<td></td>
<td>jump</td>
</tr>
</tbody>
</table>

Boolean Exprs for Controller

Op 0-5 are really Instruction bits 26-31
Func 0-5 are really Instruction bits 0-5

\[
\begin{align*}
\text{rtype} &= \neg\text{op}_5 \cdot \neg\text{op}_4 \cdot \neg\text{op}_3 \cdot \neg\text{op}_2 \cdot \neg\text{op}_1 \cdot \neg\text{op}_0, \\
\text{ori} &= \neg\text{op}_5 \cdot \neg\text{op}_4 \cdot \text{op}_3 \cdot \text{op}_2 \cdot \neg\text{op}_1 \cdot \text{op}_0 \\
\text{lw} &= \text{op}_5 \cdot \neg\text{op}_4 \cdot \neg\text{op}_3 \cdot \neg\text{op}_2 \cdot \text{op}_1 \cdot \text{op}_0 \\
\text{sw} &= \text{op}_5 \cdot \neg\text{op}_4 \cdot \text{op}_3 \cdot \neg\text{op}_2 \cdot \text{op}_1 \cdot \text{op}_0 \\
\text{beq} &= \neg\text{op}_5 \cdot \neg\text{op}_4 \cdot \neg\text{op}_3 \cdot \text{op}_2 \cdot \neg\text{op}_1 \cdot \neg\text{op}_0 \\
\text{jump} &= \neg\text{op}_5 \cdot \neg\text{op}_4 \cdot \neg\text{op}_3 \cdot \neg\text{op}_2 \cdot \text{op}_1 \cdot \neg\text{op}_0 \\
\text{add} &= \text{rtype} \cdot \text{func}_5 \cdot \neg\text{func}_4 \cdot \neg\text{func}_3 \cdot \neg\text{func}_2 \cdot \neg\text{func}_1 \cdot \neg\text{func}_0 \\
\text{sub} &= \text{rtype} \cdot \text{func}_5 \cdot \neg\text{func}_4 \cdot \text{func}_3 \cdot \neg\text{func}_2 \cdot \text{func}_1 \cdot \neg\text{func}_0 \\
\end{align*}
\]

How do we implement this in gates?
Controller Implementation

Boolean Exprs for Controller

RegDst = add + sub
ALUSrc = ori + lw + sw
MemtoReg = lw
RegWrite = add + sub + ori + lw
MemWrite = sw
nPCsel = beq
Jump = jump
ExtOp = lw + sw
ALUctr[0] = sub + beq
ALUctr[1] = ori

(assume ALUctr is 00 ADD, 01 SUB, 10 OR)

How do we implement this in gates?
Controller Implementation

```
Controller	
  Implementaton
```

```
11/7/11

Controller	
  Implementaton
```

```
Implementaton	
  Fall	
  2011	
  - Lecture	
  #30
```

```
add	
sub	
ori	
lw	
sw	
beq	
jump
```

```
RegDst	
ALUSrc	
MemtoReg	
RegWrite	
MemWrite	
nPCsel	
Jump	
ExtOp	
ALUctr[0]
ALUctr[1]
```

```
“AND” logic
```

```
“OR” logic
```

Call home, we’ve made HW/SW contact!

```
High Level Language Program (e.g., C)
```

```
Compiler
```

```
Assembly Language Program (e.g., MIPS)
```

```
Assembler
```

```
Machine Language Program (MIPS)
```

```
Machine Interpretation
```

```
Hardware Architecture Description (e.g., block diagrams)
```

```
Architecture Implementation
```

```
Logic Circuit Description (Circuit Schematic Diagrams)
```

```
High Level Language Program (e.g., C)
```

```
Compiler
```

```
Assembly Language Program (e.g., MIPS)
```

```
Assembler
```

```
Machine Language Program (MIPS)
```

```
Machine Interpretation
```

```
Hardware Architecture Description (e.g., block diagrams)
```

```
Architecture Implementation
```

```
Logic Circuit Description (Circuit Schematic Diagrams)
```

```
High Level Language Program (e.g., C)
```

```
Compiler
```

```
Assembly Language Program (e.g., MIPS)
```

```
Assembler
```

```
Machine Language Program (MIPS)
```

```
Machine Interpretation
```

```
Hardware Architecture Description (e.g., block diagrams)
```

```
Architecture Implementation
```

```
Logic Circuit Description (Circuit Schematic Diagrams)
```

```
High Level Language Program (e.g., C)
```

```
Compiler
```

```
Assembly Language Program (e.g., MIPS)
```

```
Assembler
```

```
Machine Language Program (MIPS)
```

```
Machine Interpretation
```

```
Hardware Architecture Description (e.g., block diagrams)
```

```
Architecture Implementation
```

```
Logic Circuit Description (Circuit Schematic Diagrams)
```

```
High Level Language Program (e.g., C)
```

```
Compiler
```

```
Assembly Language Program (e.g., MIPS)
```

```
Assembler
```

```
Machine Language Program (MIPS)
```

```
Machine Interpretation
```

```
Hardware Architecture Description (e.g., block diagrams)
```

```
Architecture Implementation
```

```
Logic Circuit Description (Circuit Schematic Diagrams)
```

```
High Level Language Program (e.g., C)
```

```
Compiler
```

```
Assembly Language Program (e.g., MIPS)
```

```
Assembler
```

```
Machine Language Program (MIPS)
```

```
Machine Interpretation
```

```
Hardware Architecture Description (e.g., block diagrams)
```

```
Architecture Implementation
```

```
Logic Circuit Description (Circuit Schematic Diagrams)
```

```
High Level Language Program (e.g., C)
```

```
Compiler
```

```
Assembly Language Program (e.g., MIPS)
```

```
Assembler
```

```
Machine Language Program (MIPS)
```

```
Machine Interpretation
```

```
Hardware Architecture Description (e.g., block diagrams)
```

```
Architecture Implementation
```

```
Logic Circuit Description (Circuit Schematic Diagrams)
Administrivia

• Due to time constraints – we can only allow the use of a maximum of 2 slip days on Project 4.

• Thus, while we always encourage you to get your work done on time, if you still have 3 slip days left, you may want to consider using one prior to project 4.

Review: Single-cycle Processor

• Five steps to design a processor:
  1. Analyze instruction set → datapath requirements
  2. Select set of datapath components & establish clock methodology
  3. Assemble datapath meeting the requirements
  4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.
  5. Assemble the control logic
     • Formulate Logic Equations
     • Design Circuits
### Single Cycle Performance

- Assume time for actions are
  - 100ps for register read or write; 200ps for other events

- Clock rate is?

<table>
<thead>
<tr>
<th>Instr</th>
<th>Instr fetch</th>
<th>Register read</th>
<th>ALU op</th>
<th>Memory access</th>
<th>Register write</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td>200ps</td>
<td>100 ps</td>
<td>800ps</td>
</tr>
<tr>
<td>sw</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td>200ps</td>
<td></td>
<td>700ps</td>
</tr>
<tr>
<td>R-format</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td></td>
<td>100 ps</td>
<td>600ps</td>
</tr>
<tr>
<td>beq</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td></td>
<td></td>
<td>500ps</td>
</tr>
</tbody>
</table>

- What can we do to improve clock rate?
- Will this improve performance as well?
  - Want increased clock rate to mean faster programs
Gotta Do Laundry

• Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, fold, and put away
  – Washer takes 30 minutes
  – Dryer takes 30 minutes
  – “Folder” takes 30 minutes
  – “Stasher” takes 30 minutes to put clothes into drawers

Sequential Laundry

• Sequential laundry takes 8 hours for 4 loads
Pipelined Laundry

- Pipelined laundry takes 3.5 hours for 4 loads!

Pipelining Lessons (1/2)

- Pipelining doesn’t help latency of single task, it helps throughput of entire workload
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Time to “fill” pipeline and time to “drain” it reduces speedup: 2.3X v. 4X in this example
Suppose new Washer takes 20 minutes, new Stasher takes 20 minutes. How much faster is pipeline?

- Pipeline rate limited by slowest pipeline stage
- Unbalanced lengths of pipe stages reduces speedup

**Steps in Executing MIPS**

1. **IFetch**: Instruction Fetch, Increment PC
2. **Dec**: Instruction Decode, Read Registers
3. **Exec**: 
   - Mem-ref: Calculate Address 
   - Arith-log: Perform Operation
4. **Mem**: 
   - Load: Read Data from Memory 
   - Store: Write Data to Memory
5. **WB**: Write Data Back to Register
1. Instruction Fetch
2. Decode/Register Read
3. Execute
4. Memory
5. Write Back

• Need registers between stages
  – To hold information produced in previous cycle
More Detailed Pipeline

IF for Load, Store, ...

Chapter 4 — The Processor — 25

Chapter 4 — The Processor — 26
ID for Load, Store, ...

EX for Load
MEM for Load

WB for Load – Oops!
So, in conclusion

- You now know how to implement the control logic for the single-cycle CPU.
  - (actually, you already knew it!)

- Pipelining improves performance by increasing instruction throughput: exploits ILP
  - Executes multiple instructions in parallel
  - Each instruction has the same latency

- Next: hazards in pipelining:
  - Structure, data, control