Computer architecture

The simplest processor

Wouter M. Koolen

Advanced Topics
23-2-12
About me

- Work in machine learning ...
- .. but generally interested in most of computer science
- Fervent programmer
- Computer architecture as a hobby
- Author of SIM-PL simulator for digital hardware

http://www.science.uva.nl/amstel/SIM-PL
Source material

- Computer Organization & Design
  - David A. Patterson and John L. Hennessy
- Van 0 en 1 tot processor
  - Met simulaties van schakelingen in SIM-PL
  - Eon-Beads
What do I expect of you / What would really help

You …

▶ want to learn
▶ have seen a digital circuit (e.g. gate, adder, flip-flop).
▶ wrote a program (e.g. if, goto, increment)
▶ are not mortally afraid of bits and hexadecimal

I …
What do I expect of you / What would really help

You ...
- want to learn
- have seen a digital circuit (e.g. gate, adder, flip-flop).
- wrote a program (e.g. if, goto, increment)
- are not mortally afraid of bits and hexadecimal

I ...
- will provide trick questions to guide your thinking

And most importantly ...
What do I expect of you / What would really help

You …

▶ want to learn
▶ have seen a digital circuit (e.g. gate, adder, flip-flop).
▶ wrote a program (e.g. if, goto, increment)
▶ are not mortally afraid of bits and hexadecimal

I …

▶ will provide trick questions to guide your thinking

And most importantly …

You raise your hand when you get lost
Goal of this lecture

I present the simplest processor

Understanding

- hardware design
  - You can do it too
  - Baseline for more complex designs
  - Many (esoteric) designs found niches
- execution of software
  - Programming (e.g. embedded devices)
  - Compiler architecture
  - The why of hardware eccentricities
Goal of this lecture
Pierce layers of abstraction

Transistor
  └── Gate
      ├── Stateful circuit
      │     └── Memory
      └── Boolean circuit
          └── Arithmetic circuit
              └── Calculator
                  └── Processor
                      └── Assembly language
                          └── C
                              └── Assembler
                                  └── Compiler

Simple Processor
Koolen
Introduction
Calculator
Immediates
Jumps
Data memory
Procedure calls
Conclusion
Advanced tricks
Calculator

Goal

- 16 variables (memory cells)
- program (list of instructions)
- 4 operations
Calculator

Goal

- 16 variables (memory cells)
- program (list of instructions)
- 4 operations

Instruction set
ADD, SUB, AND, COPY
Calculator

Goal

- 16 variables (memory cells)
- program (list of instructions)
- 4 operations

Instruction set
ADD, SUB, AND, COPY

Example program

0: ADD $6, $3, $4 Set Reg6 to Reg3 + Reg4
1: SUB $7, $3, $4 Set Reg7 to Reg3 − Reg4
2: COPY $8, $6 Set Reg8 to Reg6
Implementation: the hardware circuit
The driving force

Purpose: Synchronise operation of components

- Usually some kind of oscillating crystal

- High and low levels

- Positive (up) edge and negative (down) edge

- A *clock cycle* is: up – high – down – low
Program counter

Purpose: maintain the current position in the program

- One address
- Increment triggered by positive clock edge
Instruction memory

Purpose: store the program

- 16 bit address
- 14 bit data
Registers

Purpose: maintain the state of program variables

- 16 registers named $0$, $1$, $\ldots$, $15$
- 16 bits each
- Two read ports
- One write port (triggered by negative clock edge)
Arithmetic logic unit (ALU)

Purpose: performs computation

<table>
<thead>
<tr>
<th>$S_0$</th>
<th>$S_1$</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>$A + B$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>$A - B$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>$A &amp; B$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>$B$</td>
</tr>
</tbody>
</table>

- 2 bits to select operation
Calculator: the circuit

[Diagram of Calculator circuit with nodes labeled as follows:
- Program Counter
- Instruction Memory
- Registers (First register, Second register, Destination reg.)
- ALU
- Clock]
In action

Say $3$ initially contains 7, while $4$ contains 5.

0: ADD $6$, $3$, $4$  Set Reg6 to Reg3 + Reg4
1: SUB $7$, $3$, $4$  Set Reg7 to Reg3 - Reg4
2: COPY $8$, $6$  Set Reg8 to Reg6
Say $3$ initially contains 7, while $4$ contains 5.

0: ADD $6$, $3$, $4$  Set Reg6 to Reg3 + Reg4
1: SUB $7$, $3$, $4$  Set Reg7 to Reg3 - Reg4
2: COPY $8$, $6$  Set Reg8 to Reg6
In action

Say $3$ initially contains 7, while $4$ contains 5.

0: ADD $6$, $3$, $4$ Set Reg6 to Reg3 + Reg4
1: SUB $7$, $3$, $4$ Set Reg7 to Reg3 - Reg4
2: COPY $8$, $6$ Set Reg8 to Reg6
In action

Say $3$ initially contains 7, while $4$ contains 5.

0: ADD $6$, $3$, $4$  Set Reg6 to Reg3 + Reg4
1: SUB $7$, $3$, $4$  Set Reg7 to Reg3 - Reg4
2: COPY $8$, $6$  Set Reg8 to Reg6

![Diagram of ALU with registers and program counter]
Summary

Calculator

- 16 memory cells
- 4 operations
- Executes program (list of instructions) sequentially
- Timing
Why does an instruction take 14 bits of memory?

Is COPY $1, $1 safe?

What about ADD $1, $1, $1?

Can we increment a register? How?

Can we multiply 2 registers? How?

Can we execute an if statement? How?
Immediates

Goal
Use of immediates (constants) in instructions
Immediates

Goal
Use of immediates (constants) in instructions

Instruction set
ADD, SUB, AND, COPY, **ADDI, SUBI, ANDI, LOADI**
Immediates

Goal
Use of immediates (constants) in instructions

Instruction set
ADD, SUB, AND, COPY, ADDI, SUBI, ANDI, LOADI

Example program
0: LOADI $1, 0x3000 Load $3000_{hex} in Reg1
1: LOADI $2, 0x2000 Load $2000_{hex} in Reg2
2: SUB $3, $1, $2 Set Reg3 to Reg1 - Reg2
3: ADDI $4, $3, 0x200 Set Reg4 to Reg3 + $0200_{hex}
Implementing immediates

Do we use an immediate?
Immediate value
Multiplexer

Purpose: channel chooser

- 1 bit to select which input is passed on
Immediates in action

0:  LOADI $1, 0x3000  Load $3000_{hex}$ in Reg1
1:  LOADI $2, 0x2000  Load $2000_{hex}$ in Reg2
2:  SUB $3, $1, $2  Set Reg3 to Reg1 - Reg2
3:  ADDI $4, $3, 0x200  Set Reg4 to Reg3 + 0200_{hex}
Immediates in action

0: LOADI $1, 0x3000  
Load $3000_{\text{hex}}$ in Reg1

1: LOADI $2, 0x2000  
Load $2000_{\text{hex}}$ in Reg2

2: SUB $3, $1, $2  
Set Reg3 to Reg1 - Reg2

3: ADDI $4, $3, 0x200  
Set Reg4 to Reg3 + $0200_{\text{hex}}$

```
0: LOADI $1, 0x3000  
Load 3000_{\text{hex}} in Reg1

1: LOADI $2, 0x2000  
Load 2000_{\text{hex}} in Reg2

2: SUB $3, $1, $2  
Set Reg3 to Reg1 - Reg2

3: ADDI $4, $3, 0x200  
Set Reg4 to Reg3 + 0200_{\text{hex}}
```
Immediates in action

0: LOADI $1, 0x3000  Load 3000\textsubscript{hex} in Reg1
1: LOADI $2, 0x2000  Load 2000\textsubscript{hex} in Reg2
2: SUB $3, $1, $2  Set Reg3 to Reg1 - Reg2
3: ADDI $4, $3, 0x200  Set Reg4 to Reg3 + 0200\textsubscript{hex}
Immediates in action

0: LOADI $1, 0x3000  Load $3000_{hex}$ in Reg1
1: LOADI $2, 0x2000  Load $2000_{hex}$ in Reg2
2: SUB $3, $1, $2   Set Reg3 to Reg1 - Reg2
3: ADDI $4, $3, 0x200 Set Reg4 to Reg3 + 0200_{hex}
Immediates in action

0: \text{LOADI} \; \$1, \; 0x3000 \quad \text{Load } 3000_{\text{hex}} \text{ in Reg1}
1: \text{LOADI} \; \$2, \; 0x2000 \quad \text{Load } 2000_{\text{hex}} \text{ in Reg2}
2: \text{SUB} \; \$3, \; \$1, \; \$2 \quad \text{Set Reg3 to Reg1} - \text{Reg2}
3: \text{ADDI} \; \$4, \; \$3, \; 0x200 \quad \text{Set Reg4 to Reg3} + \; 0200_{\text{hex}}
Summary

- A little more hardware ...
- ... to allow immediates as last argument
- Immediates part of instruction
How many bits do we need for each instruction?
Can we increment a register? How?
Can we execute `ADDII $1, 0x1, 0x2`? How?
Can we multiply 2 registers? How?
Can we execute an `if` statement? How?
Conditional execution and jumps

Goal
Implement if/else, switch, for, while, goto ...
Conditional execution and jumps

Goal
Implement if/else, switch, for, while, goto ...

Instruction set
ADD(I), SUB(I), AND(I), COPY, LOADI, BRA, BZ, BEQ
Conditional execution and jumps

Goal
Implement if/else, switch, for, while, goto ...

Instruction set
ADD(I), SUB(I), AND(I), COPY, LOADI, BRA, BZ, BEQ

Example program

```plaintext
  LOADI  a, 8    # a = 8;
  LOADI  b, 4    # b = 4;
  LOADI  r, 0    # r = 0;
loop:
  BZ   b, end    # while (b != 0) {
     ADD  r, r, a  #   r += a;
  SUBI  b, b, 1  #     --b;
  BRA   loop     # }
end:
```

Implementing jumps

Do we jump?
Where do we jump to?
Do we write to register?
Arithmetic logic unit (ALU)

Purpose: performs computation and tests

<table>
<thead>
<tr>
<th>$S_0$</th>
<th>$S_1$</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>$A + B$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>$A - B$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>$A &amp; B$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>$B$</td>
</tr>
</tbody>
</table>

- 2 bits to select operation
- Zero bit is set when output is zero
Summary

- A bunch more hardware
- Hardware executes a branch when
  - Instruction is a branch instruction
  - Test succeeds (ALU outputs zero)
- Target of jump is encoded in instruction
(Trick) questions

- How many bits do we need for each instruction?
- How can we test for (with A and B registers)
  
  \[ A = 0 \quad A = 1 \quad A = B \quad A \neq B \quad A < B \]

- Can we multiply 2 registers? How?
- Can we execute an if statement? How?
- Can we execute an if/else statement? How?
- Is Branch the negation of RegWrite?
More/main memory

Goal
Add memory. We add $2^{16} = 65536$ variables
More/main memory

Goal
Add memory. We add $2^{16} = 65536$ variables

Instruction set
ADD(I), SUB(I), AND(I), COPY, LOADI, BRA, BZ, BEQ, LW, SW
More/main memory

Goal
Add memory. We add $2^{16} = 65536$ variables

Instruction set
ADD(I), SUB(I), AND(I), COPY, LOADI, BRA, BZ, BEQ, LW, SW

Example program
LW $1, 10, $6  # Load mem[Reg6 + 10] into Reg1
SW $2, 10, $6  # Store $2 into mem[Reg6 + 10]
Implementing main memory

Do we read from/write to memory? Write what where? Value read?
Summary

- We simply “bolted on” some memory
- Both in hardware
- And in software
(Trick) questions

► How many bits do we need for each instruction?
► Why not simply increase the number of registers?
► Can we perform a computation (say $A + B$) and write the result to memory using a single instruction?
► Can we execute an if statement? How?
► What if we want more than $2^{16} = 65536$ variables?
► Can a program modify itself? (polymorphic code)
Procedure calls

Goal
Re-use blocks of code

Example program
LOADI $arg1, 1
LOADI $arg2, 2
LOADI $arg3, 3
CALL $ra, Add3

#--------------- Add3 procedure -----------------
Add3:
ADD $val1, $arg1, $arg2
ADD $val1, $val1, $arg3
RETURN $ra
Procedure calls

Goal
Re-use blocks of code

Instruction set
ADD(I), SUB(I), AND(I), COPY, LOADI, BRA, BZ, BEQ, LW, SW, CALL, RETURN

Example program
LOADI $arg1, 1
LOADI $arg2, 2
LOADI $arg3, 3
CALL $ra, Add3

#---------- Add3 procedure ----------
Add3:
ADD $val1, $arg1, $arg2
ADD $val1, $val1, $arg3
RETURN $ra
Implementing procedure calls

Do we CALL/RETURN?
Store PC+1 in register (for CALL)
Load PC from register (for RETURN)
Summary

- We allow store and load of PC
- Increment to return to instruction *after* call
- Contract (calling convention) between caller and callee
(Trick) questions

- How can we compute the PC at a given instruction?
- Can we implement a dispatch table? (function pointer)
- Can a procedure call another procedure?
- What about recursion?
Conclusion

- We built a general purpose processor
- In incremental steps
Advanced tricks

- Asynchronous design
  *No clock*

- Caching
  *Fast small memory on top of slow big memory*

- Register stacks
  *Accelerated procedure calls*

- Floating point arithmetic, multimedia, encryption
  *Upgrade the ALU*

- Very large instruction word (VLIW)
  *Multiple independent ALUs*

- Pipelining
  *Execute multiple (sub-)instructions simultaneously*

- Multi-core/processor
  *Multiple processors attached to a single main memory*