07 - Program Flow Control

Andreas Ehliar

August 31, 2015
The lecture on thursday needs to move
The current computer lab (Bussen) is pretty nice since it has dual monitors.

However, the computers does not have enough memory to comfortably run matlab and firefox at the same time.

Modelsim and firefox running at the same time will probably not improve things.

Question: Do we move to a lab with faster computers but only one monitor per computer?
Control path introduction

Instruction decoder

Program flow controller

Processor configuration

Flags and status

PC finite state machine

Program memory

Instruction decoder

Data path

Addressing path

Peripherals

Control path itself

[Liu2008]
Quick hint for lab 3:
- You might want to refresh your memory regarding Moore and Mealy-style state machines before embarking on lab 3.
- (You will need to create a Mealy-style FSM there.)
Jobs allocated in the control path

- Supplies the right instruction to execute
  - Normal next PC, Branches, Call/return and loops
- Decodes instructions into control signals
  - For data path, control path, memory addressing, and peripherals/bus
- Special control for DSP
  - Loop controller
Instruction decode - Registered and non-registered

[Liu2008]

▶ Try to keep as many control signals registered as possible
▶ Control signals dealing with instruction fetch (branches, loop control, etc) might be unregistered for performance reasons.
Two techniques for instruction decoding: Centralized vs distributed

Program memory

Instruction decoder

- Combinational control signals
- Control signals stage x
- Control signals stage x+1
- Control signals stage x+2
- Control signals stage x+3

Program memory

Instruction decoder

- Combinational control signals
- Control signals stage x
- Control signals stage x+1
- Control signals stage x+2
And now, for a complete control path example

- A very simplified processor
  - The execution unit contains a simple arithmetic unit
  - 16 general purpose registers (16 bits each)
  - 7 instructions: Four arithmetic, 3 branches
  - 8 bit address space for the program memory
### Instruction set and binary coding

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD rD,rS,rT</td>
<td>0000 ssss tttt dddd</td>
</tr>
<tr>
<td>SUB rD,rS,rT</td>
<td>0001 ssss tttt dddd</td>
</tr>
<tr>
<td>CMP rD,rS,rT</td>
<td>0010 ssss tttt 0000</td>
</tr>
<tr>
<td>MUL rD,rS,rT</td>
<td>0011 ssss tttt dddd</td>
</tr>
<tr>
<td>JMP A</td>
<td>0100 0000 aaaa aaaa</td>
</tr>
<tr>
<td>JMP.EQ A</td>
<td>0101 0000 aaaa aaaa</td>
</tr>
<tr>
<td>JMP.NE A</td>
<td>0110 0000 aaaa aaaa</td>
</tr>
</tbody>
</table>

▶ **Question:** Why should bit 3:0 of the CMP instruction be 0000 rather than don’t care? What about bit 11:8 of the branch instructions?

▶ (After all, a don’t care here will simplify the instruction decoder)
It is always a good idea to leave some space for future instructions.

It is a good idea to trap illegal instructions to an exception, allowing emulation of such instructions (although this is slow!).

However, in some cases we may want to create an instruction decoder that handles certain bits as don’t care, to improve the clock frequency (more on this later).

(The rest of this example assumes that some bits are don’t care for simplicity though.)
Instruction set and binary coding

Mnemonic | Encoding
--- | ---
ADD rD,rS,rT | 0000 ssss tttt dddd
SUB rD,rS,rT | 0001 ssss tttt dddd
CMP rD,rS,rT | 0010 ssss tttt 0000
MUL rD,rS,rT | 0011 ssss tttt dddd
JMP A | 0100 0000 aaaa aaaa
JMP.EQ A | 0101 0000 aaaa aaaa
JMP.NE A | 0110 0000 aaaa aaaa

▶ Side question: What is missing to make this instruction set *minimally* useful?
Instruction set and binary coding

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD rD,rS,rT</td>
<td>0000 ssss tttt dddd</td>
</tr>
<tr>
<td>SUB rD,rS,rT</td>
<td>0001 ssss tttt dddd</td>
</tr>
<tr>
<td>CMP rD,rS,rT</td>
<td>0010 ssss tttt 0000</td>
</tr>
<tr>
<td>MUL rD,rS,rT</td>
<td>0011 ssss tttt dddd</td>
</tr>
<tr>
<td>JMP A</td>
<td>0100 0000 aaaa aaaa</td>
</tr>
<tr>
<td>JMP.EQ A</td>
<td>0101 0000 aaaa aaaa</td>
</tr>
<tr>
<td>JMP.NE A</td>
<td>0110 0000 aaaa aaaa</td>
</tr>
</tbody>
</table>

Side question: What is missing to make this instruction set *minimally* useful?

Answer: I/O and some way to load constants into registers (e.g. immediate arguments)
Our execution unit

![Diagram of execution unit](image-url)
The complete datapath

Register File

Execution Unit

ctrl_rfaaddr
Address A
Data A
OPA
16
ctrl_alu
ctrl_update_flag
ctrl_mux

ctrl_rfbaddr
Address B
Write Enable
Address W
Data W
OPB
16
ctrl_rfwe
ctrl_rfwaddr
16
// Not so hard...
ctrl_rfaaddr = de_insn[11:8];
ctrl_rfbaddr = de_insn[7:4];
always @* begin
  // Default statements to avoid
  // latches. (Very important!)
  ctrl_alu = 0;
  ctrl_mux = 0;
  ctrl_update_flag = 0;

  // Note that we are checking
  // ex_insn here, not de_insn
  case(ex_insn[15:12])
  4'b0000: begin // ADD
      ctrl_alu = 0;
      ctrl_mux = 0;
      ctrl_update_flag = 1;
    end

end
Instruction decoding
Arithmetic instructions - Execute Stage

4'b0001: begin // SUB
    ctrl_alu = 1;
    ctrl_mux = 0;
    ctrl_update_flag = 1;
end

4'b0010: begin // CMP
    ctrl_alu = 1;
    ctrl_mux = 0;
    ctrl_update_flag = 1;
end

4'b0011: begin // MUL
    ctrl_mux = 1;
end
endcase
end
Instruction decoding
Arithmetic instructions - Writeback Stage

```verilog
// Instruction decoder writeback stage
always @* begin
    ctrl_rfwe = 0;
    ctrl_rfwbaddr=wb_insn[3:0];
    case(wb_insn[15:12])
        // ADD
        4'b0000: ctrl_rfwe = 1;
        // SUB
        4'b0001: ctrl_rfwe = 1;
        // MUL
        4'b0011: ctrl_rfwe = 1;
    endcase
end
```

Andreas Ehliar
07 - Program Flow Control
Instruction decoding
Unconditional jump

// Control signals, decoder stage
// Only a limited amount of control signals should be generated
// combinationally here.
always @* begin
    jumpaddr = de_insn[7:0];
    ctrl_jump_uncond = 0;
    case(de_insn[15:12])
        4'b0100: begin // JMP
            ctrl_jump_uncond = 1;
            end
        endcase
end
Instruction decoding

The problem with jumps

- Consider the following program:
  - `jmp 0x59`
  - `add r5, r2, r3`
- The add is already being fetched when the jump is decoded
Option 1 - Don’t use pipelining
  ▶ Bonus: If you don’t need any performance in your system you don’t need to pass TSEA26...

Option 2 - Discard the extra instruction
  ▶ Not very good for performance...
Instruction decoding
Handling control hazards

- Option 3 - Consider it a "feature"
  - The add is executed in the delay slot of the jump
  - This is very common for simple RISC-like processors
- Option 4 - Use branch prediction to avoid the problem
  - Not really a part of this course
What about conditional jumps?

```
CMP r0, r5
JMP.EQ 0x57
```

The flag is available late in the pipeline.

Andreas Ehliar
07 - Program Flow Control
Program Counter with support for conditional jumps

jumpaddr  conditional_jumpaddr

ctrl_jump_checkflag

ctrl_jump_uncond

ctrl_jump_checkflag

ctrl_jump_mode

alu_zflag

Program memory

16

Instruction decoder

Combinational control signals from the decoder

16

dex_insn

Instruction decoder

Control signals for execution stage

16

ex_insn

Instruction decoder

Control signals stage for writeback stage

16

wb_insn

Instruction decoder

Andreas Ehliar

07 - Program Flow Control
// Control signals, execute stage
always @* begin
    ctrl_jump_checkflag = 0;
    ctrl_jump_mode = 0;
    case(ex_insn[15:12])
        4’b0101: begin // JMP.EQ
            ctrl_jump_checkflag = 1;
            ctrl_jump_mode = 1;
        end
        4’b0110: begin // JMP.NE
            ctrl_jump_checkflag = 1;
            ctrl_jump_mode = 0;
        end
    endcase
end
Program Counter with support for conditional jumps

- Two delay slots for conditional jumps
  - In a real processor the flags will probably be available even later in the pipeline
- Ways to avoid this - Predict not taken
  - Always start instructions after branch
  - Flush the pipeline if the flag test is negative
    - For arithmetic instructions this can be done by disabling writeback
- Slightly more advanced
  - Use a bit in the instruction word to predict taken/not-taken
Great! We have solved all problems. Or…?

- Are there any other problems?
Data hazards

- Consider the following instruction sequence
  
  add \( r0, r1, r2 \)
  
  add \( r4, r0, r3 \)

Old \( r0 \)

New \( r0 \)
Handling data hazards

- One solution - "This is also a feature"
  - Also known as "the lazy solution"
  - Can actually be a real feature in some way since it allows you to use the pipeline registers as temporary storage
    - Don’t do this if you can avoid it!
    - I did. I regretted it just a year later when I wanted to add interrupts...
  - Better variant: Consider this undefined behavior
    - Simulator or assembler disallows code like this (e.g. srsim)
Handling data hazards

- Stall the pipeline
  - Stop the pipeline above the decode stage
  - Let the decode stage insert NOP instructions until the result is ready.
Handling data hazards

- Register forwarding (also known as register bypass)
- Bypass register file using muxes
- Most elegant solution
- Could limit clockrate
- Not possible to do in all cases
  - Notably memories and other instructions with long pipelines
Structural hazards

- If two resources are used at the same time
- Example to the right
  - Memory access pipeline is one clock cycle longer than ALU

```
load r0,[r1]
add r2,r3,r4
```
Dealing with structural hazards

- The usual suspects: Stall or simply consider it a "feature"
- Another solution: add more hardware to simply avoid the problem
  - Example: Extra write-port on the register file
  - Example: Extra forwarding paths
  - Drawback: Can be very expensive
Pipeline hazards summary

- Control hazard
  - Cannot determine jump address and/or jump condition early enough
- Data hazard
  - An instruction is not finished by the time an instruction tries to access the result (or possibly, write a new result)
- Structural hazard
  - Two instructions tries to utilize the same unit at the same time from different locations in the pipeline
Diminishing returns when adding pipeline stages

[Liu2008]
The instruction decoder handles timing critical signals first in an optimistic fashion.

- Will make verification harder! (More corner cases)
Instruction decoder tricks

- Other ways
  - Ignore the (hopefully slight) performance hit. *(Recommended if at all possible.)*
  - Trust users never to use “undefined” instructions (Hah!)
  - If you use an instruction cache: change undefined instructions into specific “trap” instructions. *(This is simple if all instructions are the same length, impossible otherwise (in the general case).)*
Predecoding can also help in other cases

- A few extra bits in the instruction cache (or instruction word) can be beneficial for other cases
- Conditional/unconditional branches
- Hazard detection
Instruction encoding problems

- Goal: As many instructions in as few bits as possible
- Challenges
  - Space for future expansion (look at x86 for a scary example...)
  - Space for immediate data (including jump addresses)
  - Should be easy for the instruction decoder to parse
Instruction encoding problems

- Immediate data
  - Alternative 1: Enough space for native data width
  - Alternative 2: Not wide enough. Need two instructions to set a register to a constant (sethi/setlo)
Instruction encoding problems

- Branch target address
  - Relative addressing (saves bits, typically enough)
  - Absolute addressing (probably required for unconditional branches and subroutine calls)
The program counter module

From I-decoder
From register file
From stack
To stack
Code source
Boot

Boot FSM
PC FSM
Loop controller
Program memory
Instruction decoder
Instruction

To register file
To ALU
To MAC
To AGU
To memories
Immediate data

Write enable
Boot data
Boot address
PC
start
finish

[LIU2008]
PC FSM

Default state: PC <= PC + 1

PC <= 0
PC <= Interrupt service entry
PC <= Stack pop
PC <= PC
PC <= Jump target address

Exception
In loop
Hold reset
Accept interrupt
reset
reset
reset
reset
reset
reset
reset
reset

DefaultTo loop
PC <= PC
reset
reset

reset

[ Liu2008 ]
PC Example

[Liu2008]

Andreas Ehliar

07 - Program Flow Control
What kind of jumps do we need?

- **Absolute**
  - $PC = \text{Immediate from instruction word}$
  - $PC = \text{REG}$ (Note: used for function pointers!)

- **Relative**
  - $PC = PC + \text{Immediate}$
  - $PC = PC + \text{REG}$ (Necessary for PIC (Position independent code))
Loop controller

[Li2008]
Loop controller

Number of iterations

Number of instructions

[Liu2008]
Handling subroutines

- Return address can be pushed to
  - Special call/return stack in PC FSM
    - Example: Small embedded processors (e.g. PIC12/PIC16)
  - Normal memory
    - CISC-like general purpose processors (e.g. 68000, x86)
  - Register
    - RISC-like processors (e.g. MIPS, ARM)
    - Up to the subroutine to save the return address if another subroutine call is made
PC with hardware stack

[M1 <= 1 IF Push & SPR[2:0]=000;
M2<=1 IF Push & SPR[2:0]=001;
M3<=1 IF Push & SPR[2:0]=010;
M4<=1 IF Push & SPR[2:0]=011;
Else M1 <= M2 <= M3 <= M4<=0;

Overflow <= Push & SPR [2]
Underflow <= Pop & SPR=000
OpError <= Pop & Push

[Liu2008]
Dealing with interrupts

- Desirable features from the user:
  - Low latency
  - Configurable priority for different interrupt sources

- Desirable features from the hardware designer
  - Easy to verify
Handling low latency interrupts

- Save only PC and Status register
  - Interrupt handlers must be written to use as few registers as possible to avoid having to save/restore such registers
- Save many registers in hardware
  - Convenient for programmer
  - More complex hardware/interrupt handling
- Shadow registers
  - A processor with 16 user visible registers (r0-r15) may actually have 24 registers in the register file.
  - r0-r7 is replaced by r16-r23 during an interrupt
Handling low latency interrupts

- **Reserved registers**
  - Certain registers are reserved for the interrupt handler and may not be used by regular programs
  - See MIPS ABI
  - More generally, this can be done in GCC if you are careful
    - `register int interrupt_handler_reserved asm ("r5");`
    - All code needs to be recompiled with this declaration visible!
Reducing verification time

- Disallow interrupts at certain times
  - Typically branch delay slots
  - Introduces jitter in interrupt response
  - Can be handled by introducing a delay in interrupt-handling when handling interrupts happening outside delay slots
Interrupts in delay slots

▶ WARNING: Ensure that the following kind of code doesn’t hang your processor:

```Assembly
loop:
    jump ds3 loop
    nop
    nop
    nop
```
Interrupts in delay slots

- Disallow interrupts at certain times
  - What about the following?

```asm
loop:
    jump ds3 loop
    jump ds3 loop ; Typically not allowed by
    nop ; the specification, but you
    nop ; probably don’t want code
    nop ; like this to hang the system.
    ; (See the Cyrix COMA bug for
    ;   a similar example.)
```