# 07 - Program Flow Control

### Andreas Ehliar

September 23, 2014

#### 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



[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.)

- 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



### [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



- 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

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

- 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
  - Allows 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.)

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

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

### Our execution unit



## The complete datapath



### Control Path (first version)



# Instruction decoding Arithmetic instructions - RF readout



# Instruction decoding Arithmetic instructions - Execute Stage



# 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

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



# 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



# Instruction decoding Handling control hazards

- 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?





The flag is available late in the pipeline

### 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

#### Are there any other problems?

 Consider the following instruction sequence add r0,r1,r2 add r4,r0,r3



### 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)

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

- 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



- 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



- 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

### 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



Andreas Ehliar 07 - Program Flow Control

 The instruction decoder handles timing critical signals first in an optimistic fashion



Will make verification harder! (More corner cases)

### 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

- 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

#### 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)

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



[Liu2008]

## PC FSM





[Liu2008]

#### Absolute

- PC = Immediate from instruction word
- PC = REG (Note: used for function pointers!)
- Relative
  - $\blacktriangleright PC = PC + Immediate$
  - PC = PC + REG (Necessary for PIC (Position independent code))



## PC FSM supporting loop instruction





#### 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



[Liu2008]

- Desirable features from the user:
  - Low latency
  - Configurable priority for different interrupt sources
- Desirable features from the hardware designer
  - Easy to verify

- 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

#### 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!

- 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

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

```
loop:
jump ds3 loop
nop
nop
nop
```

### Disallow interrupts at certain times

What about the following?

#### 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.)