CS61C : Machine Structures

Lecture #19 – Designing a Single-Cycle CPU

2007-7-26

Scott Beamer

Instructor

AI Focuses on Poker

nytimes.com

Review

- **N-bit adder-subtractor done using N 1-bit adders with XOR gates on input**
  - **XOR serves as conditional inverter**
- **CPU design involves Datapath,Control**
  - **Datapath in MIPS involves 5 CPU stages**
  - **1) Instruction Fetch**
  - **2) Instruction Decode & Register Read**
  - **3) ALU (Execute)**
  - **4) Memory**
 - **5) Register Write**

Datapath Summary

- **The datapath based on data transfers required to perform instructions**
- **A *controller* causes the right transfers to happen**
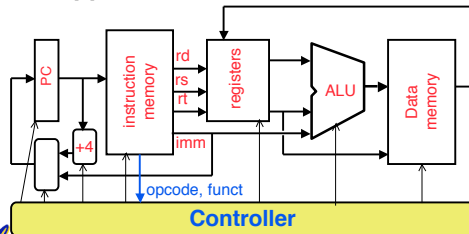
CPU clocking (1/2)

For each instruction, how do we control the flow of information though the datapath?

- **Single Cycle CPU: All stages of an instruction are completed within one *long* clock cycle.**
  - **The clock cycle is made sufficient long to allow each instruction to complete all stages without interruption and within one cycle.**
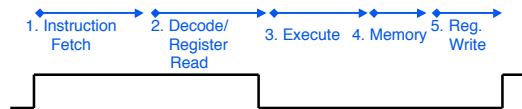
CPU clocking (2/2)

For each instruction, how do we control the flow of information though the datapath?

- **Multiple-cycle CPU: Only one stage of instruction per clock cycle.**
  - **The clock is made as long as the slowest stage.**
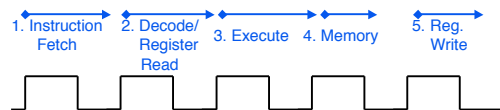
Several significant advantages over single cycle execution: Unused stages in a particular instruction can be skipped OR instructions can be pipelined (overlapped).

How to Design a Processor: step-by-step

- **1. Analyze instruction set architecture (ISA) ⇒ datapath requirements**
  - **meaning of each instruction is given by the *register transfers***
  - **datapath must include storage element for ISA registers**
  - **datapath must support each register transfer**
- **2. Select set of datapath components and establish clocking methodology**
- **3. Assemble datapath meeting requirements**
- **4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.**
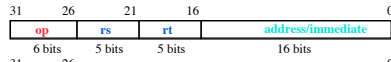- **5. Assemble the control logic (hard part!)**

Review: The MIPS Instruction Formats

• **All MIPS instructions are 32 bits long. 3 formats:**

• **R-type**

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

31 26 21 16 11 6 0

• **I-type**

op	rs	rt	address/immediate
6 bits	5 bits	5 bits	16 bits

31 26 21 16 0

• **J-type**

op	target address
6 bits	26 bits

31 26 0

• **The different fields are:**
  • **op:** operation ("opcode") of the instruction
  • **rs, rt, rd:** the source and destination register specifiers
  • **shamt:** shift amount
  • **funct:** selects the variant of the operation in the "op" field
  • **address / immediate:** address offset or immediate value
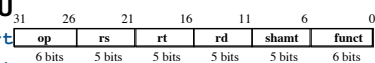 • **target address:** target address of jump instruction

---

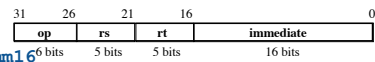## Step 1a: The MIPS-lite Subset for today
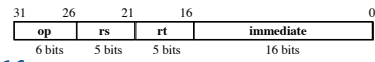
• **ADDU and SUBU**
  • `addu rd,rs,rt`
  • `subu rd,rs,rt`

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

31 26 21 16 11 6 0

• **OR Immediate:**
  • `ori rt,rs,imm16`

op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

31 26 21 16 0

• **LOAD and STORE Word**
  • `lw rt,rs,imm16`
  • `sw rt,rs,imm16`

op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

31 26 21 16 0

• **BRANCH:**
  • `beq rs,rt,imm16`

op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

31 26 21 16 0
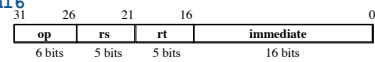
Register Transfer Language

• **RTL gives the _meaning_ of the instructions**

{op , rs , rt , rd , shamt , funct} ← MEM[PC]

{op , rs , rt , Imm16} ← MEM[PC]

• **All start by fetching the instruction**

inst	Register Transfers	
ADDU	R[rd] ← R[rs] + R[rt];	PC ← PC + 4
SUBU	R[rd] ← R[rs] – R[rt];	PC ← PC + 4
ORI	R[rt] ← R[rs]	zero_ext(Imm16);	PC ← PC + 4
LOAD	R[rt] ← MEM[R[rs] + sign_ext(Imm16)];	PC ← PC + 4
STORE	MEM[R[rs] + sign_ext(Imm16)] ← R[rt];	PC ← PC + 4
BEQ	if (R[rs] == R[rt]) then	
	PC ← PC + 4 + (sign_ext(Imm16) || 00)	
	else PC ← PC + 4	

Step 1: Requirements of the Instruction Set

• **Memory (MEM)**
  • **instructions & data (will use one for each)**
• **Registers (R: 32 x 32)**
  • **read RS**
  • **read RT**
  • **Write RT or RD**
• **PC**
• **Extender (sign/zero extend)**
• **Add/Sub/OR unit for operation on register(s) or extended immediate**
• **Add 4 or extended immediate to PC**
• **Compare registers?**
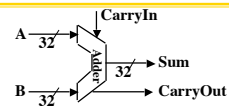

Step 2: Components of the Datapath

• **Combinational Elements**

• **Storage Elements**
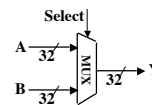 • **Clocking methodology**

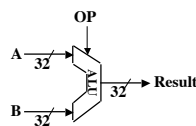## Combinational Logic Elements (Building Blocks)

• **Adder**

• **MUX**

• **ALU**

ALU Needs for MIPS-lite + Rest of MIPS

- **Addition, subtraction, logical OR, ==:**

```
ADDU R[rd] = R[rs] + R[rt]; ...

SUBU R[rd] = R[rs] – R[rt]; ...

ORI  R[rt] = R[rs] | zero_ext(Imm16)...

BEQ  if ( R[rs] == R[rt] )...
```

- **Test to see if output == 0 for any ALU operation gives == test. How?**

- **P&H also adds AND, Set Less Than (1 if A < B, 0 otherwise)**

- **ALU follows chap 5**

What Hardware Is Needed? (1/2)

- **PC: a register which keeps track of memory addr of the next instruction**

- **General Purpose Registers**
  - **used in Stages 2 (Read) and 5 (Write)**
  - **MIPS has 32 of these**

- **Memory**
  - **used in Stages 1 (Fetch) and 4 (R/W)**
  - **cache system makes these two stages as fast as the others, on average**


What Hardware Is Needed? (2/2)

- **ALU**
  - **used in Stage 3**
  - **something that performs all necessary functions: arithmetic, logicals, etc.**
  - **we'll design details later**

- **Miscellaneous Registers**
  - **In implementations with only one stage per clock cycle, registers are inserted between stages to hold intermediate data and control signals as they travels from stage to stage.**
  - **Note: Register is a general purpose term meaning something that stores bits. Not all registers are in the "register file".**


Storage Element: Idealized Memory

- **Memory (idealized)**
  - **One input bus: Data In**
  - **One output bus: Data Out**
- **Memory word is selected by:**
  - **Address selects the word to put on Data Out**
  - **Write Enable = 1: address selects the memory word to be written via the Data In bus**
- **Clock input (CLK)**
  - **The CLK input is a factor ONLY during write operation**
  - **During read operation, behaves as a combinational logic block:**
   - **Address valid ⟹ Data Out valid after "access time."**
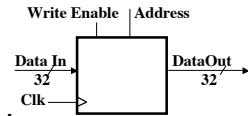
Storage Element: Register (Building Block)

- **Similar to D Flip Flop except**
  - **N-bit input and output**
  - **Write Enable input**
- **Write Enable:**
  - **negated (or deasserted) (0): Data Out will not change**
  - **asserted (1): Data Out will become Data In on positive edge of clock**
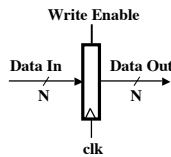
Storage Element: Register File

- **Register File consists of 32 registers:**
  - **Two 32-bit output busses: busA and busB**
  - **One 32-bit input bus: busW**
- **Register is selected by:**
  - **RA (number) selects the register to put on busA (data)**
  - **RB (number) selects the register to put on busB (data)**
  - **RW (number) selects the register to be written via busW (data) when Write Enable is 1**
- **Clock input (clk)**
  - **The clk input is a factor ONLY during write operation**
  - **During read operation, behaves as a combinational logic block:**
    - **RA or RB valid ⟹ busA or busB valid after "access time."**
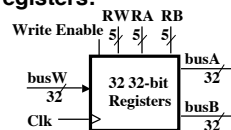
Administrivia

- **Assignments**
  - HW5 due Tonight
  - HW6 due 7/29
- **Midterm**
  - Grading standards up
  - If you wish to have a problem regraded
    - Staple your reasons to the front of the exam
    - Return your exam to your TA
- **Scott is now holding regular OH on Fridays 11-12 in 329 Soda**

CS61C L19 CPU Design : Designing a Single-Cycle CPU (19)

Beamer, Summer 2007 © UCB

Step 3: Assemble DataPath meeting requirements

- **Register Transfer <u>Requirements</u>**
  ⇒ Datapath <u>Assembly</u>
- **Instruction Fetch**
- **Read Operands and Execute Operation**

CS61C L19 CPU Design : Designing a Single-Cycle CPU (20)

Beamer, Summer 2007 © UCB
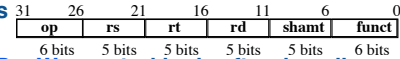

3a: Overview of the Instruction Fetch Unit

- **The common RTL operations**
  - Fetch the Instruction: mem[PC]
  - Update the program counter:
    - Sequential Code: PC ← PC + 4
    - Branch and Jump: PC ← "something else"

CS61C L19 CPU Design : Designing a Single-Cycle CPU (21)

Beamer, Summer 2007 © UCB
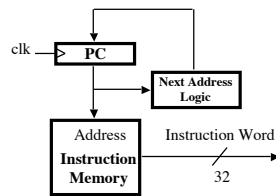
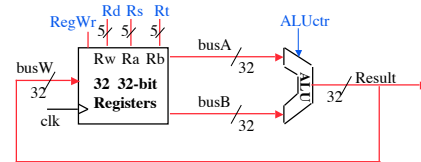
3b: Add & Subtract

- **R[rd] = R[rs] op R[rt]   Ex.: `addU rd,rs,rt`**
  - Ra, Rb, and Rw come from instruction's Rs, Rt, and Rd fields

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

  31   26 21   16   11   6   0

  - ALUctr and RegWr: control logic after decoding the instruction

Already defined the register file & ALU

CS61C L19 CPU Design : Designing a Single-Cycle CPU (22)

Beamer, Summer 2007 © UCB

Clocking Methodology

- **Storage elements clocked by same edge**
- **Being physical devices, flip-flops (FF) and combinational logic have some delays**
  - Gates: delay from input change to output change
  - Signals at FF D input must be stable before active clock edge to allow signal to travel within the FF (set-up time), and we have the usual clock-to-Q delay
- **"Critical path" (longest path through logic) determines length of clock period**
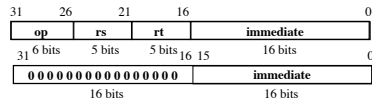
CS61C L19 CPU Design : Designing a Single-Cycle CPU (23)

Beamer, Summer 2007 © UCB

Register-Register Timing: One complete cycle

CS61C L19 CPU Design : Designing a Single-Cycle CPU (24)

Beamer, Summer 2007 © UCB

3c: Logical Operations with Immediate
- R[**rt**] = R[rs] op ZeroExt[imm16]]

31	26	21	16	0
op	rs	rt	immediate	

6 bits · 5 bits · 5 bits · 16 bits

31	16	15	0
0000000000000000		immediate	

16 bits · 16 bits
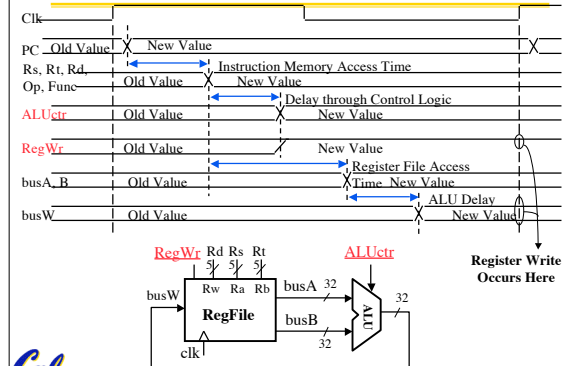
But we're writing to Rt register??

RegWr Rd Rs Rt

ALUctr

busW

RegFile

busA 32

busB

ALU

32

32

clk

CS61C L19 CPU Design : Designing a Single-Cycle CPU (25)   Beamer, Summer 2007 © UCB

3c: Logical Operations with Immediate
- R[**rt**] = R[rs] op ZeroExt[imm16]]

31	26	21	16	0
op	rs	rt	immediate	

6 bits · 5 bits · 5 bits · 16 bits

31	16	15	0
0000000000000000		immediate	

16 bits · 16 bits

RegDst   Rd Rt

1   0
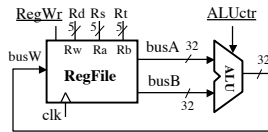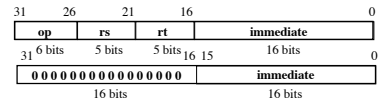
What about Rt register read??

RegWr   Rs Rt

32

RegFile   busA 32

busB

ALUctr

ALU

32

clk

imm16   16   ZeroExt   32

ALUSrc
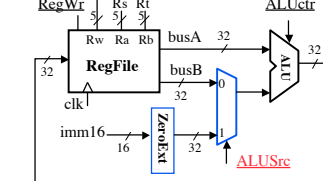
- Already defined 32-bit MUX; Zero Ext?

CS61C L19 CPU Design : Designing a Single-Cycle CPU (26)   Beamer, Summer 2007 © UCB

3d: Load Operations
- R[**rt**] = Mem[R[rs] + SignExt[imm16]]
  Example: `lw rt,rs,imm16`

31	26	21	16	0
op	rs	rt	immediate	

6 bits · 5 bits · 5 bits · 16 bits

RegDst   Rd Rt

1   0

RegWr   Rs Rt

32

Rw Ra Rb

RegFile   busA 32

busB

ALUctr

ALU

32

clk

imm16   16   ZeroExt   32

ALUSrc

CS61C L19 CPU Design : Designing a Single-Cycle CPU (27)   Beamer, Summer 2007 © UCB

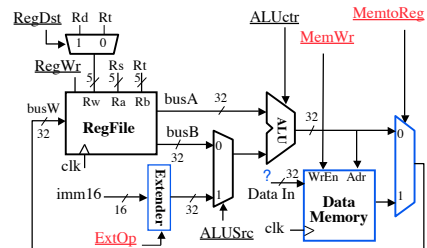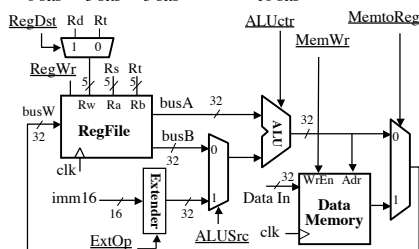
3d: Load Operations
- R[**rt**] = Mem[R[rs] + SignExt[imm16]]
  Example: `lw rt,rs,imm16`

31	26	21	16	0
op	rs	rt	immediate	

6 bits · 5 bits · 5 bits · 16 bits

RegDst   Rd Rt

1   0

ALUctr

MemtoReg

RegWr   Rs Rt

busW

32

Rw Ra Rb

RegFile   busA 32

busB

ALU

32

MemWr

0

clk

imm16   16   Extender   32

? 32   WrEn Adr

Data In

Data Memory

0

1

ExtOp   ALUSrc   clk

CS61C L19 CPU Design : Designing a Single-Cycle CPU (28)   Beamer, Summer 2007 © UCB

3e: Store Operations
- Mem[R[rs] + SignExt[imm16]] = R[rt]
  Ex.: `sw rt, rs, imm16`

31	26	21	16	0
op	rs	rt	immediate	

6 bits · 5 bits · 5 bits · 16 bits

RegDst   Rd Rt

1   0

ALUctr

MemtoReg

MemWr

RegWr   Rs Rt

busW

32

Rw Ra Rb

RegFile   busA 32

busB

ALU

32

0

0

clk

imm16   16   Extender   32

32   WrEn Adr

Data In

Data Memory

1

ExtOp   ALUSrc   clk

CS61C L19 CPU Design : Designing a Single-Cycle CPU (29)   Beamer, Summer 2007 © UCB

3e: Store Operations
- Mem[R[rs] + SignExt[imm16]] = R[rt]
  Ex.: `sw rt, rs, imm16`

31	26	21	16	0
op	rs	rt	immediate	

6 bits · 5 bits · 5 bits · 16 bits

RegDst   Rd Rt

1   0

ALUctr

MemtoReg

RegWr   Rs Rt

MemWr

busW

32

Rw Ra Rb

RegFile   busA 32

busB

ALU

32

0

0

clk

imm16   16   Extender   32

32   WrEn Adr

Data In

Data Memory

1

ExtOp   ALUSrc   clk

CS61C L19 CPU Design : Designing a Single-Cycle CPU (30)   Beamer, Summer 2007 © UCB

3f: The Branch Instruction

31	26	21	16		0
op	rs	rt	immediate		
6 bits	5 bits	5 bits	16 bits		
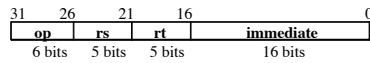
beq rs, rt, imm16

- mem[PC] Fetch the instruction from memory
- Equal = R[rs] == R[rt] Calculate branch condition
- if (Equal) Calculate the next instruction's address
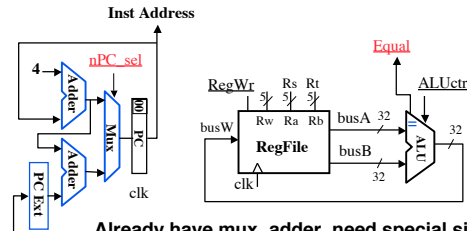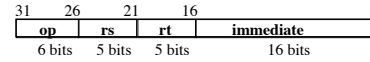 - PC = PC + 4 + (SignExt(imm16) x 4)
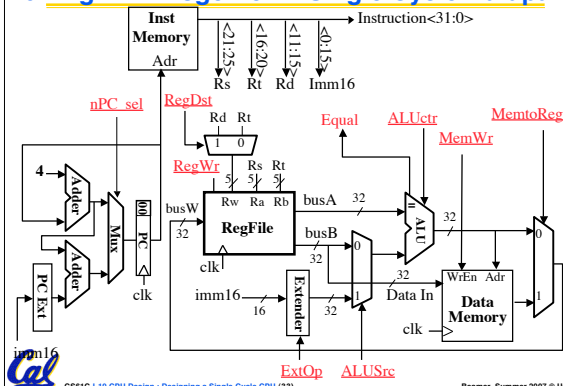 - else
  - PC = PC + 4

Datapath for Branch Operations

- **beq rs, rt, imm16**
Datapath generates condition (equal)

31	26	21	16		0
op	rs	rt	immediate		
6 bits	5 bits	5 bits	16 bits		

Inst Address

Already have mux, adder, need special sign extender for PC, need equal compare (sub?)

Putting it All Together:A Single Cycle Datapath

Peer Instruction

A. For the CPU designed so far, the Controller only needs to look at **opcode/funct** and **Equal**

B. Adding **jal** would only require changing the **Instruction Fetch** block

C. Making our single-cycle CPU multi-cycle will be **easy**

	ABC
0:	FFF
1:	FFT
2:	FTF
3:	FTT
4:	TFF
5:	TFT
6:	TTF
7:	TTT

How to Design a Processor: step-by-step

1. **Analyze instruction set architecture (ISA) => datapath requirements**
   - meaning of each instruction is given by the *register transfers*
   - datapath must include storage element for ISA registers
   - datapath must support each register transfer
2. **Select set of datapath components and establish clocking methodology**
3. **Assemble datapath meeting requirements**
4. **Analyze implementation of each instruction to determine setting of control points that effects the register transfer.**

Assemble the control logic