

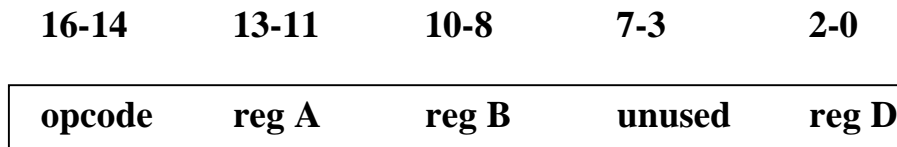
Datapath and Control (Ch 5)

Path from source to object

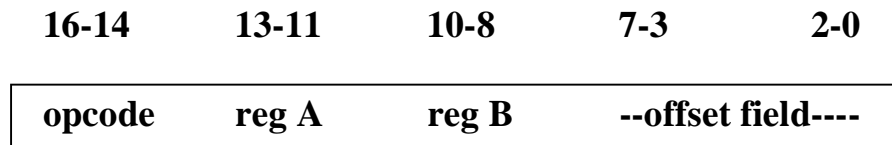
- source file (.c file)
- compiler (.s file)
- assembler (.o file)
- linker (stitch together individual modules)
- object file (a.out file)
 - header, text, data, relocation info, symbol table, debugging info
- loader (from disk to memory)
- memory layout

LC-99

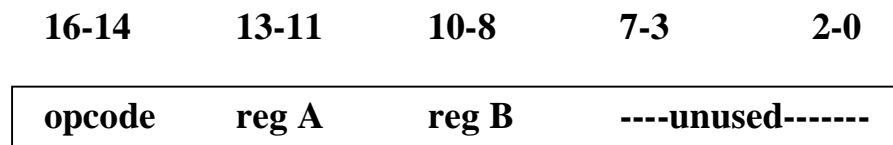
- R-type (add, nand)



- I-type (lw, sw, beq)



- J-type (jalr)



- O-type (halt, noop)

16-14 13-11 10-8 7-3 2-0

opcode	-----unused-----
--------	------------------

CPU organization

- Datapath and control
 - how to decide the datapath?
 - What resources do we need in the datapath?
- A simple datapath LC-99
 - PC, IR, register file
 - MUXes at address inputs to register file, why?
 - optional MUXes at ALU inputs, why?
 - ALU
 - optional buffers at input and output of ALU
 - optional buffers at outputs of register file

CPU registers

- IR, PC
 - where does each get its data input?
 - how do we use each of its contents?
 - control signals needed to actuate each?
- GPR or register file
 - external interface?
 - internal logic?
 - data input?
 - load vs. arithmetic instructions
 - control signals needed?

- Why do we need muxes at ALU input?
 - ALU
 - arithmetic instructions
 - where are the operands in LC-99?
 - address calculations
 - what instructions need this? where are the operands?
 - PC management
 - what is involved in this?
- Why buffer at output of register file?
- Why buffer at output of ALU?

Bus-based Datapath

- data bus
- address bus
- units electrically connected to the bus
 - what does that mean?
 - how to ensure only one driving the bus at a time?
- how to make better use of the address bus?
 - use it for both data and address
 - eliminates need for buffering register file output
 - introduces the need for latching ALU output

Control Design

- What is involved in instruction execution?
 - fetch, decode, execute, fetch operands, store operands
- designing the control unit
 - given instruction set, and DP
 - write flowcharts: this is no different from writing C code!
 - adopt a control regime
 - hardwired
 - » FSM
 - microprogrammed control
 - » stored program

Hardwired Control

- What states do we need in the FSM?
 - Actions during instruction fetch - ifetch1
 - want to init fetch and increment PC
 - PC to address bus: how to effect this?
 - Read memory: how to effect this?
 - increment PC: how to effect this?
 - code snippet to simulate the above

```
machine.control.PCdriveAddr = ON;  
machine.control.RdMem = ON;  
machine.control.latchALU = ON;  
machine.control.ALUop = ADDop;  
machine.control.ALUC = ALUCone;
```

– Actions during instruction fetch - ifetch2

- latch memory data (instruction) into IR: how?
- code snippet to simulate this

```
machine.control.EnableMem = ON;
```

```
machine.control.latchIR = ON;
```

– Actions during instruction fetch - ifetch3

- write the incremented value into PC: how? Why not in ifetch2?
- code snippet to simulate this

```
machine.control.ALUdriveData = ON;
```

```
machine.control.PCWr = ON;
```

– Actions during instruction fetch - ifetch3 contd.

- Decode instruction: how?
- Code snippet to simulate this

```
opcode = (machine.instReg >> 14) & 0x7;
```

```
regA = (machine.instReg >> 11) & 0x7;
```

```
regB = (machine.instReg >> 8) & 0x7;
```

```
regD = machine.instReg & 0x7;
```

```
offsetValue = machine.instReg & 0xff;;
```

```
/* Now choose next state.
```

```
* As you construct the states below,
```

```
* try to share
```

```
* states among instructions when possible.
```

```
*/
```

```
if (opcode == ADD){.....
```

- what next?
 - 4 different FSM sequences depending on instruction type
 - R-type, I-type, J-type, O-type

- R-type FSM
 - select regA and regB from register file: how?
 - ALU op (ADD or NAND); store result in ALUresult: how?
 - write ALUout into register file at regD: how?
 - can all of the above be done in one cycle?
- Where do we go from here?

- I-type FSM: load/store
 - address arithmetic: regA plus offset
 - result into ALUresult
 - ALUresult to address bus
 - load
 - read memory onto data bus
 - latch into regB
 - store
 - drive regB onto the data bus
 - back to F1

- I-type FSM: BEQ
 - subtract regA and regB
 - if equal
 - latch 0 into cond register (if *all* ALU bits are 0)
 - latch 1 into cond register (if *any* ALU bit is a 1)
 - if cond register is 0 then
 - then PC+offset to ALUresult
 - where did the +1 go?
 - ALUresult to PC
 - else nothing
 - back to F1

- J-type FSM: JALR
 - PC to register regB specified in IR
 - contents of register regA in IR to PC
 - back to F1

- O-type FSM: HALT
 - Quiesce state
- O-type FSM: NOP
 - back to F1