April 5

- Best to start on that assignment...
- Read Section 6.1
- Concrete FSM example
- How instructions execute

Recognizing Numbers

Recognize the regular expression for floating point numbers

‘ ’* [‘+’ ‘-’]? [‘0’ - ‘9’] * (‘.’ [‘0’ - ‘9’] *)? (e [‘+’ ‘-’]? [‘0’ – ‘9’]+)?

Examples:

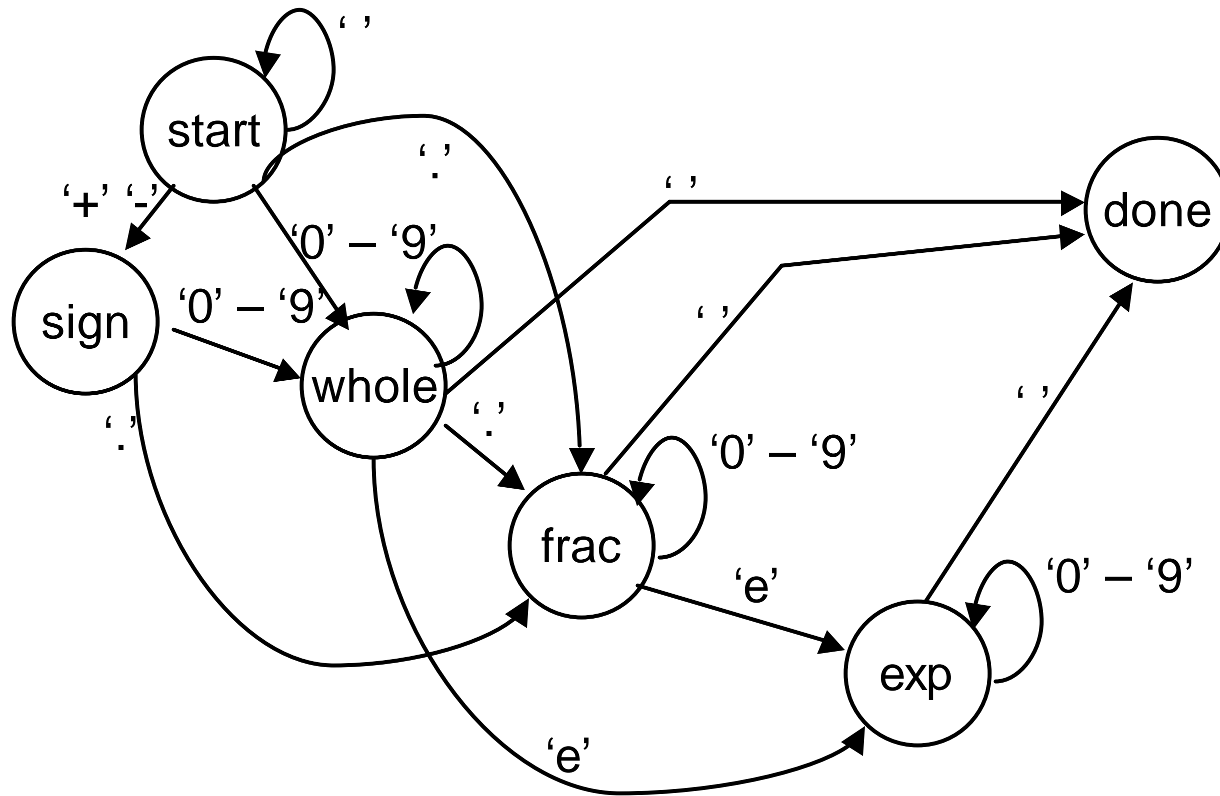
+123.456e23

.456

    1.5e-10

-123

FSM Diagram



FSM Table

IN : STATE ✍ NEW STATE

‘ ’ : start ✍ start

‘0’ | ‘1’ | ... | ‘9’ : start ✍ whole

‘+’ | ‘-’ : start ✍ sign

‘.’ : start ✍ frac

‘0’ | ‘1’ | ... | ‘9’ : sign ✍ whole

‘.’ : sign ✍ frac

‘0’ | ‘1’ | ... | ‘9’ : whole ✍ whole

‘.’ : whole ✍ frac

‘ ’ : whole ✍ done

‘e’ : whole ✍ exp

‘e’ : frac ✍ exp

‘0’ | ‘1’ | ... | ‘9’ : frac ✍ frac

‘ ’ : frac ✍ done

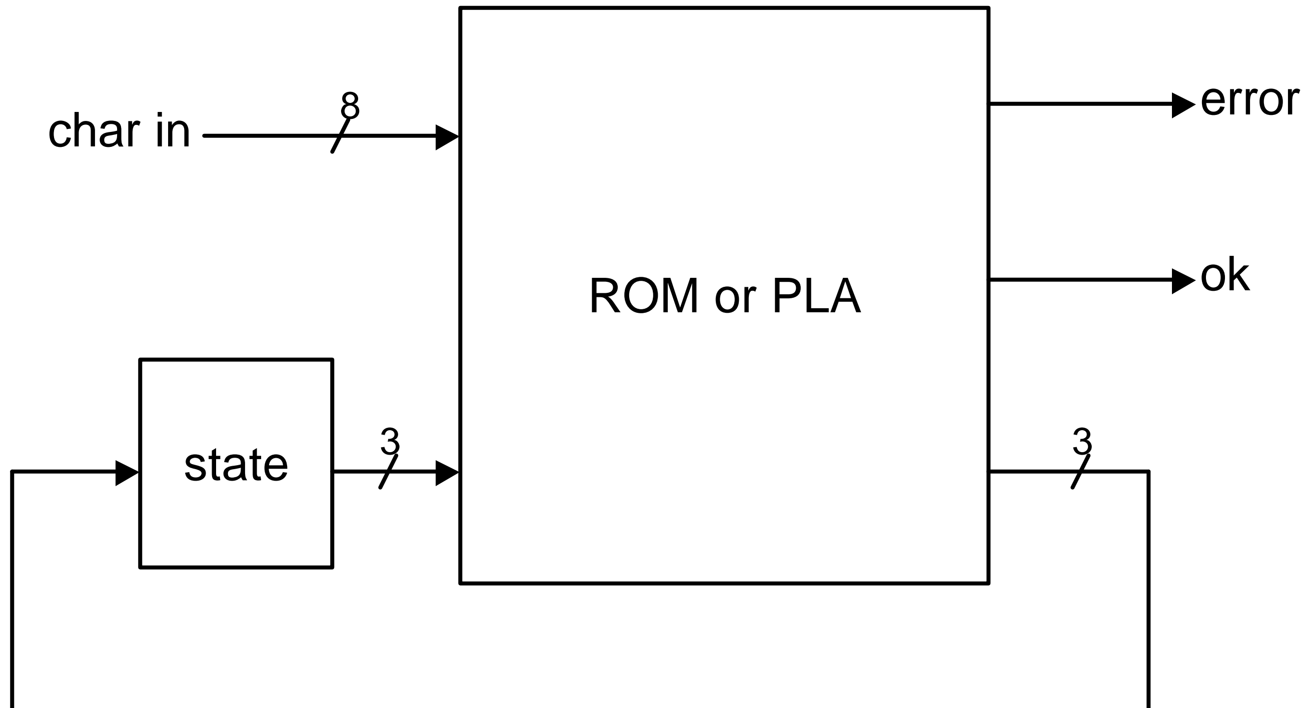
‘0’ | ‘1’ | ... | ‘9’ : exp ✍ exp

‘ ’ : exp ✍ done

STATE ASSIGNMENTS

```
start = 0 = 000
sign  = 1 = 001
whole = 2 = 010
frac  = 3 = 011
exp   = 4 = 100
done  = 5 = 101
error = 6 = 110
```

FSM Implementation



Our PLA has:

- 11 inputs
- 5 outputs

FSM Take Home

- With ***JUST*** a register and some logic, we can implement complicated sequential functions like recognizing a FP number.
- This is useful in its own right for compilers, input routines, etc.
- The reason we're looking at it here is to see how designers implement the complicated sequences of events required to implement instructions
- Think of the OP-CODE as playing the role of the input character in the recognizer. The character AND the state determine the next state (and action).

Five Execution Steps

- Instruction Fetch
- Instruction Decode and Register Fetch
- Execution, Memory Address Computation, or Branch Completion
- Memory Access or R-type instruction completion
- Memory Read Completion

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!

- A FSM looks at the op-code to determine how many...

Step 1: Instruction Fetch

- Use PC to get instruction and put it in the Instruction Register.
- Increment the PC by 4 and put the result back in the PC.
- Can be described succinctly using RTL "Register-Transfer Language"

```
     IR = Memory[PC];
I    PC = PC + 4;
```

IR is “Instruction Register”

What is the advantage of updating the PC now?

Step 2: Instruction Decode and Register Fetch

- Read registers rs and rt in case we need them
- Compute the branch address in case the instruction is a branch
- RTL:

```
A = Reg[IR[25-21]];
B = Reg[IR[20-16]];
ALUOut = PC + (sign-extend(IR[15-0]) << 2);
```

- We aren't setting any control lines based on the instruction type
  (we are busy "decoding" it in our control logic)

Step 3 (instruction dependent)

- ALU is performing one of three functions, based on instruction type
- Memory Reference:

```
ALUOut = A + sign-extend(IR[15-0]);
```

- R-type:

```
ALUOut = A op B;
```

- Branch:

```
if (A==B) PC = ALUOut;
```

Step 4 (R-type or memory-access)

- Loads and stores access memory

```
MDR = Memory[ALUOut];
      or
Memory[ALUOut] = B;
```

MDR is Memory Data Register

- R-type instructions finish

```
Reg[IR[15-11]] = ALUOut;
```

Step 5 Memory Read Completion

- `Reg[IR[20-16]]= MDR;`

Summary:

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR = Memory[PC] PC = PC + 4			
Instruction decode/register fetch	A = Reg [IR[25-21]] B = Reg [IR[20-16]] ALUOut = PC + (sign-extend (IR[15-0]) << 2)			
Execution, address computation, branch/ jump completion	ALUOut = A op B	ALUOut = A + sign-extend (IR[15-0])	if (A ==B) then PC = ALUOut	PC = PC [31-28] II (IR[25-0]<<2)
Memory access or R-type completion	Reg [IR[15-11]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory [ALUOut] = B		
Memory read completion		Load: Reg[IR[20-16]] = MDR		