# ELEE 204 Designing a Simple CPU

## Big Issues

CPU/memory speed mismatch - CPUs are fast and memory is slow. We won't deal with this issue in our architecture, but it pervades real CPU design.

Compiler designers are awesome - The compiler has to perform very sophisticated and complex tasks to translate our high-level code to its efficient assembler language form.

Assembler language design is a series of trade-offs - Programmers want larger sizes and more options. Hardware designers want smaller and simpler.

Fetch/Execute cycle - A CPU constantly fetches the next instruction from memory and executes that instruction. Again, and again, and again, and again, ...

Variable size vs. fixed size - Humans tend to operate with variable size data. Computers have fixed sizes of registers, bundles of wires, etc. In general it is simpler and cleaner for computers to use fixed size data, even though that wastes some space.

## Terminology

Data path - The parts of the CPU design where data is stored, moved or processed. Examples are the registers, the ALU, the bus and memory.

Control - The part of the CPU that turns on control signals in the right sequence based on the assembler instruction being processed.

MAR - Memory Address Register - The contents are the address that tells memory where to read or write.

op code - An unsigned integer telling what instruction this bit string represents.

PC - Program Counter - A register that holds the address of the next instruction to fetch.

IP - Instruction Pointer - Exactly the same as the PC.

IR - Instruction Register - A special register inside the CPU that holds the binary encoding of the instruction just fetched and currently being executed.

Fetch Cycle - The FSM that controls the CPU goes through exactly the same states at the beginning of each instruction. These are the states that fetch the next instruction.

## Design Version 1 (April 10)

The data path is in a separate image document.

We started with a couple simple assembler instructions to illustrate the concepts. Here are the instructions and the cycles/control signal settings to execute those instructions. For each, the choice of registers to use is arbitrary.

ADD R1, R2, R3 meaning: R1 = R2 + R3

| | |
|---|---|
| Cycle 1 | R_R2, W_OP1 |
| Cycle 2 | R_R3, W_OP2 |
| Cycle 3 | ALU_OP = "add" |
| Cycle 4 | R_ALU, W_R1 |

LOAD R1, address  meaning: get the 8b value at memory 'address' and put it in R1
in the cycles below, we assume the address is in R2

| | |
|---|---|
| Cycle 1 | W_MAR, R_R2 |
| Cycle 2 | R_MEM |
| Cycle 3 | W_R1, MEM_TO_BUS |

STORE R1, address  meaning: take 8b value from R1 and put it at memory 'address'
in the cycles below, we assume the address is in R2

| | |
|---|---|
| Cycle 1 | W_MAR, R_R2 |
| Cycle 2 | R_R1, BUS_TO_MEM |
| Cycle 3 | W_MEM |

## Assembler Language Design

Each assembler instruction will be converted into a string of bits. The series of bit strings that represent a program (the compiled version of the program) will be stored in memory. The bit strings will be multiples of 8 bits long (because memory is segmented into bytes).

There are many design decisions to be made:

- number of instructions - 16 (so 4b of op code)
- number of registers - 4 (so 2b to represent a specific register)
- addressing modes - 3 (register, memory direct, immediate)
- number of operands - 3 (for ALU type operations)
- length of bit string to represent an assembler instruction - 16b

Our assembler instructions:

- ADD R1, R2, R3  R1 = R2 + R3
- LOAD R1, address  R1 = memory[address]
- STORE R1, address  memory[address] = R1
- ADDI R1, R2, number  R1 = R2 + number encoded in the instruction
- AND R1, R2, R3  R1 = R2 AND R3
- NOT R1, R2  R1 = NOT(R2)
- JUMP address  the next instruction to execute is at memory[address]
- BRANCH.EQ address
  if the EQ bit is true, the next instruction is at memory[address]
  if not, the next instruction is right after this one in memory
- INPUT R1, device number  read I/O device 'device number' and store result in R1
- OUTPUT R1, device number  write contents of R1 to I/O device 'device number'
- COMPARE.EQ R1, R2  if R1=R2 then set EQ=1
  else set EQ=0

Possible addressing modes:

- register - the data you want is in a register
- memory direct - the data you want is in memory, so you specify the memory address

- immediate - the data you want is in the bit string representing the instruction
- memory indirect - the data you want is in memory, the memory address is in a register

## **Design Version 2** (April 15)

The data path is in a separate image document.

Binary encodings:

- ADD R1, R2, R3 op code 0000
- LOAD R1, address op code 0001
- STORE R1, address op code 0010
- ADDI R1, R2, number op code 0011
- AND R1, R2, R3 op code 0100
- NOT R1, R2 op code 0101
- JUMP address op code 0110
- BRANCH.EQ address op code 0111
- INPUT R1, device number op code 1000
- OUTPUT R1, device number op code 1001
- COMPARE.EQ R1, R2 op code 1010

For instructions that use registers, the next 2 bits after the op code are the destination register, the next 2 bits are operand register 1, and the next 2 bits (first 2 bits of the second byte) are the operand register 2. For instructions that have an address/number/device number, that is an 8b value which is the 2nd byte of the instruction.

The fetch process is:

| | |
|---|---|
| Cycle 1 | R_PC, W_MAR |
| Cycle 2 | R_MEM, MEM_TO_BUS, W_IR1, INC_PC |
| Cycle 3 | R_PC, W_MAR |
| Cycle 4 | R_MEM, MEM_TO_BUS, W_IR2, INC_PC |

The fetch is repeated at the beginning of every instruction.

ADD

| | |
|---|---|
| Cycle 5 (add) | R_REG, WHICH_REG = IR1[1..0], W_OP1 |
| Cycle 6 (add) | R_REG, WHICH_REG = IR2[7..6], W_OP2 |
| Cycle 7 (add) | ALU_OP = “add” |
| Cycle 8 (add) | R_ALU, W_REG, WHICH_REG = IR1[3..2] |

AND

| | |
|---|---|
| Cycle 5 (and) | R_REG, WHICH_REG = IR1[1..0], W_OP1 |
| Cycle 6 (and) | R_REG, WHICH_REG = IR2[7..6], W_OP2 |
| Cycle 7 (and) | ALU_OP = “and” |
| Cycle 8 (and) | R_ALU, W_REG, WHICH_REG = IR1[3..2] |

NOT

Cycle 5 (not) R_REG, WHICH_REG = IR1[1..0], W_OP1
Cycle 6 (not) ALU_OP = “not”
Cycle 7 (not) R_ALU, W_REG, WHICH_REG = IR1[3..2]

ADDI

Cycle 5 (addi) R_REG, WHICH_REG = IR1[1..0], W_OP1
Cycle 6 (addi) R_IR2, W_OP2
Cycle 7 (addi) ALU_OP = “add”
Cycle 8 (addi) R_ALU, W_REG, WHICH_REG = IR1[3..2]

LOAD

Cycle 5 (load) R_IR2, W_MAR
Cycle 6 (load) R_MEM, MEM_TO_BUS, W_REG, WHICH_REG = IR1[3..2]

STORE

Cycle 5 (store) R_IR2, W_MAR
Cycle 6 (store) R_REG, WHICH_REG=IR1[1..0], BUS_TO_MEM, W_MEM

COMPARE.EQ

Cycle 5 (compare.eq) R_REG, WHICH_REG = IR1[1..0], W_OP1
Cycle 6 (compare.eq) R_REG, WHICH_REG = IR2[7..6], W_OP2
Cycle 7 (compare.eq) ALU_OP = “subtract”
Cycle 8 (compare.eq) if ALU_result is 0, EQ=1, else EQ=0

JUMP

Cycle 5 (jump) R_IR2, W_PC

BRANCH.EQ

Cycle 5 (branch.eq) if EQ=1, then R_IR2, W_PC

INPUT

Cycle 5 (input) R_IR2, W_DEV_NUM
Cycle 6 (input) W_REG, WHICH_REG=IR1[3..2], R_IO, IO_TO_BUS

OUTPUT

Cycle 5 (output) R_IR2, W_DEV_NUM
Cycle 6 (output) R_REG, WHICH_REG=IR1[1..0], W_IO, BUS_TO_IO

At the end of the last cycle for every instruction, the control unit circles back to Cycle 1 (the fetch process).

## Control Unit

The data paths of a CPU are controlled by a group of signals, many of them controlling tri-state buffers. The cycles defined above specify which signals should be turned on for each cycle (and by default all other signals are turned off). The control unit needs to step through an FSM where the first 4 states are always the same (the fetch cycle) and the next 1, 2, 3 or 4 states depend on what instruction we are executing.

Stepping through states - We use a counter and a decoder to generate a 1-out-of-8 code to specify which cycle we are in (cycle 1, 2, 3, ...). When we reach the end of an FSM branch (the last execute cycle for an instruction), we want to start over at Cycle 1. We start over by clearing the counter to 0 (Cycle 1), and then counting up again. So the CLR signal on the counter has a big OR gate in front of it. Inputs to the OR gate are ADD8, AND8, NOT7, ADDI8, LOAD6, STORE6, COMPAREEQ8, JUMP5, BRANCHEQ5, INPUT6 and OUTPUT 6 (the last cycle of every instruction).

Cycle1, Cycle2, Cycle3 and Cycle4 are the same every time. I.e. they do not depend on what instruction it is. Cycles 5 through 8 do depend on what instruction it is.

We use AND gates to figure out what instruction it is. I.e. we use AND gates to interpret the op code. Bits IR1[7..4] are the op code. For example, applying NOT to each op code bit and using those four (NOTed) bits as input to an AND gate indicate an ADD instruction (the op code for ADD is 0000). So there is an AND gate for each instruction, all using the same 4 op code bits. Only one of those ANDs will be active at a time - the one corresponding to the current instruction.

Tying the AND gate for the ADD instruction with the Cycle5 output of the decoder gives us the ADD5 output. I.e. if the ADD AND gate is active, and the Cycle5 output of the decoder is active, then we are in Cycle5 of the ADD instruction (ADD5). We put AND gates in for every combination of instruction op code and instruction cycle, to generate 1 active signal specifying which instruction and which cycle of that instruction. So we have ADD5, ADD6, ADD7, ADD8, NOT5, NOT6, NOT7, LOAD5, LOAD6, JUMP5, INPUT5, INPUT6, etc. (one for each cycle in the previous section of this document).

To activate a control signal (in the data path portion of the architecture), put an OR gate in front of it. The input to the OR gate is every line that represents a cycle where that signal is active. For example, the W_REG control signal is on when ADD8 OR AND8 OR NOT7 OR ADDI8 OR LOAD6 OR INPUT6. As another example, W_MAR is on when CYCLE1 OR CYCLE3 OR LOAD5 OR STORE5. So every signal in the data path has a corresponding OR gate, allowing multiple cycles to turn on that signal.