CSE 141L Video Slides

todo: subtitle

Aaron Li; A15397198

University of California, San Diego

March 24, 2023

Table of Contents

Machine Type

- Machine Type
  - 8 registers
    - No special registers
    - Hardwired role in branching, accessing data memory, and the ALU
  - Accumulator
    - Hardwired role in branching, accessing data memory, and the ALU
  - Load-Store
    - Data must be loaded to the accumulator or a register before being used
    - Instructions to load/store to/from data memory
- Number of registers: 8 registers + accumulator

Instruction Format

- Instruction format:

8	7	6	5	4	3	2	1	0
Opcode					i	val		

Opcode:	The opcode
i:	Toggles registers vs immediates; 0 for registers, 1 for immediates
val:	The register number or the immediate

  - Not all instructions require `i` and/or `val`

Instructions

A: Value in the accumulator
val: The value referenced by the `val` and `i` fields of the instruction

Instruction	Op	Explanation
add	0	`A += val`
sub	1	`A -= val`
and	2	`A &= val`
or	3	`A |= val`
xor	4	`A ^= val`
dand	5	`A = &(A[7:0], val[7:0])`
dor	6	`A = |(A[7:0], val[7:0])`
dxor	7	`A = ^(A[7:0], val[7:0])`
sl	8	`A <<= val`
sr	9	`A >>= val`
emsw	10	Extract the output MSW/LSW for encoding
elsw	11	
dmsw	12	Extract the MSW/LSW for decoding
dlsw	13	
xp4	14	Puts the relevant bits in the accumulator,
xp2	15	including the parity bit. MSW should be in A
xp1	16	and LSW should be in *val*.
loadm	17	$A \leftarrow \mathrm{Mem}[val]$
loadv	18	$A \leftarrow val$
storem	19	$\mathrm{Mem}[val] \leftarrow A$
storev	20	$val \leftarrow A$. Writes register; behavior when `i == 0` is undefined
slt	21	`if (A < val) then` $A \leftarrow 1$ `else` $A \leftarrow 0$)
beq	22	`if (A == 0) then` relative jump by `val` instructions
rb	23	Relative jump by `val` instructions
ab	24	Absolute jump to `val << 2`
done	31	Sets the `done` flag to indicate the program has finished

Data Path Diagram

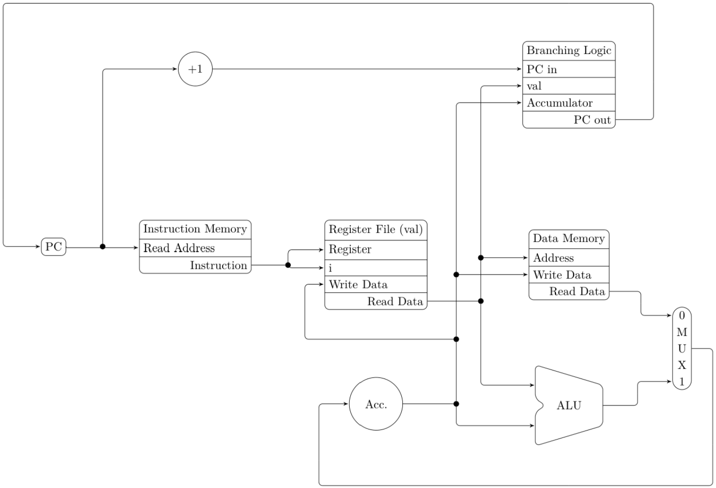

Figure: Data path Diagram (No control signals)

Block Diagram

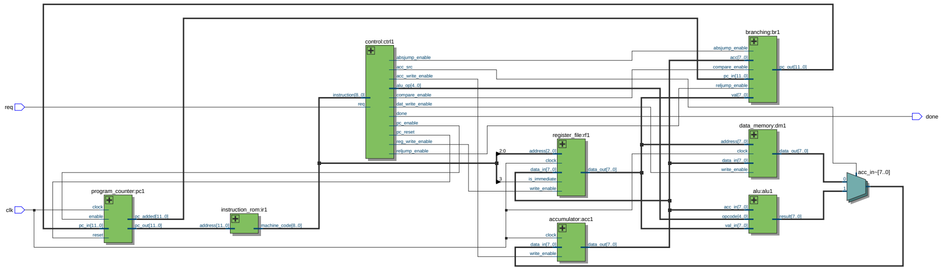

Figure: Full block diagram with control signals

- Relatively simple interactions between modules makes the design simpler
- Interesting modifications
  - Register file has to support immediates
- Custom modules
  - All except for instruction memory and data memory (Still modified)
  - PC handles adding to keep logic contained and takes a reset signal
  - Register file supports using address as immediates
  - Accumulator is just a single-value register
  - Branching handles branching using input from the PC, `val`, accumulator, and control
  - ALU handles all ALU operations and passes through the accumulator by default

Table of Contents

Assembly Syntax

- Example instructions:

```
add     i 1
xor     r 3
```

- Amount of whitespace is ignored
- Leading/trailing whitespace is ignored
- Any text starting from a # are comments is ignored
- Lines without an instruction are ignored

Explain assembly: Data

- Obtain Data
  - Small constants (0-7) can be loaded as immediates (`loadv i 7`)
  - Larger constants must be assembled from smaller immediates using various instructions such as bit shifting and `or`.
    Example for loading 30:
    ```
    loadv   i 7
    sl      i 3
    or      i 2
    ```
  - Load data from data memory with `loadm` or a register/immediate with `loadv`
- Manipulate Data
  - Data loaded in accumulator
  - Use instructions on accumulator value
    - Use register/immediates for other operand
  - Result is in the accumulator
- Store results
  - To registers: `storev`
  - To data memory: `storem`

Explain Assembly: Subroutines/Jumps

- Conditional jumps
  - `beq`: If `A == 0` then branch forward by `val` instructions
- Unconditional jumps
  - `rb`: Branch forward by `val` instructions
  - ab Branch to the `(val << 2)`th instruction. Use NOP padding if necessary.
- Other:
  - Backward conditional jump: Use `beq` to branch over instructions that absolute branch to the jump location
  - Branch over branch instructions
  - Load larger constants to a register to branch larger distances or to larger absolute addresses

Explain Assembly: Program 1

- Loop through all messages
    - Load input data
    - Set data bits in output data (in registers)
    - Extract and set parity bits
    - Store output data
- `done`

Explain Assembly: Program 2

- Based on Piazza Post @36, but modified to be simpler to implement
- Let S_8, S_4, S_2, S_1, S_2 be parity calculated from input
- for each message
  - Read message from data memory
  - Create $p = \{S_8, S_4, S_2, S_1\}$
  - If $S_0 = 0$: 0 or 2 errors
    - Extract and store LSW
    - Extract MSW
    - If $p = 0$: No errors: do nothing
    - Else ($p \neq 0$): 2 errors: set bit
    - Store MSW
  - Else: ($S_0 \neq 0$: 1 error)
    - Flip p^{th} bit in input message
    - Extract data bits to output
    - Set F_0
    - Store output
- `done`

Explain Assembly: Program 3

- Load first byte
- Loop
  - Count matches within current byte (loop)
  - Update counters in memory for occurences with byte and bytes with pattern
  - If no next byte: Update counter in memory and `done`
  - Else
    - Load next byte
    - Count occurences in byte boundary (unrolled loop)
    - Update counter in memory
    - Reset counter
    - current ← next

Table of Contents

Explain Testbench

Using the provided test benches for each program.

- Load instruction memory from file
- Randomize input data
- Compute expected output
- Set data memory to input
- Run program until `done` flag
- Compare output from data memory to computed (manually for program 3)

Proves that the program running on the simulated processor has the same result that was calculated by the testbench.

Testbench Results

- Run each test bench in Questa to show

Table of Contents

Evaluate Performance

- Accumulator machine + load-store means more instructions
- Higher instruction count for most things
  - Using larger constants require multiple instructions
  - Storing data after manipulating it requires a separate instruction
- Some annoying bit-moving operations handled by a few specialized instructions, much less than otherwise.