8 Bit Computer

Matt Ferreira and Wendy Ide
The Cooper Union
ECE 151 Computer Architecture
Professor Marano
May 9, 2016

Green Sheet (11 Instructions):

Add (acc = $r+$s)
Nand (acc = $r nand $s)
Slt (if $r<$s, acc. = 1, else acc. = 0)
Move ($r = acc)
Srl (shift accumulator right 0-7 places)
Jr ONLY FOR $ra use 11011 011
Jal
Sw (($r) = acc)
Addi (acc. = $r + immediate)
Lw (acc = ($r))
Beq (if $r=$s, PC+signed address)

NOTE: store immediates/offsets/beq (pos or neg) offset as two's complement
Store addresses in $sp, $gp as normal
Store numbits for srl in decimal so range 0 -> 7 - the hardware concatenates 00000 in front to make 8 bits, so it's always positive
Input Jal address into the assembler in decimal (direct addressing -> converts to decimal)

Opcode	Instruction	Bytes	Bit pattern	Type
00	Add	1	[2] [$r] [$s]	R
01	Nand	1	[2] [$r] [$s]	R
10	Slt	1	[2] [$r] [$s]	R
11000	Move (from acc)	1	[5] [$r]	O
11001	Srl	1	[5] [numbits(3)]	O
11010	Jrra	1	[5] [*waste(3)]	O
11011	Jal	2	[5] [bits(3)] [address(8)]	J
11100	Sw	2	[5] [$r] [$s] [offset(5)]	M
11101	Addi	2	[5] [$r] [immediate(8)]	A
11110	Lw	2	[5] [$r] [$s] [offset(5)]	M
11111	Beq	2	[5] [$r] [$s] [additional bits(2)] [$d]	B

- 8 addressable registers (8 bits wide each) (plus Ra3 register)
- In Lw and Sw, $r is source/destination, $s is base register
- Hardware will determine instruction length from opcode
- Slt sets a 1 on accumulator (if less than)
- 11 bit wide stack, extra three bits for Ra3
- $ra is 8+3 bits- sw and lw are linked
- Beq is PC relative. Must branch with two's complement 10 bits.

Memory Addressing

- Stack is BYTE addressable
- Everything else (user text) is WORD addressable. Each word is 2 bytes.
- Everything is little endian

Pseudo-instructions:

la varname -> loads the given address into the accumulator
printint varname -> loads the address into the accumulator and calls the print int syscall
printstr varname -> loads the address into the accumulator and calls the print str syscall

Registers:

Register	Dec	Bin
$0	0	000
$gp	1	001
$sp	2	010
$ra8	3	011
$r0	4	100
$r1	5	101
$r2	6	110
$r3	7	111

All registers except $sp, $gp, and $ra8 are two’s complement

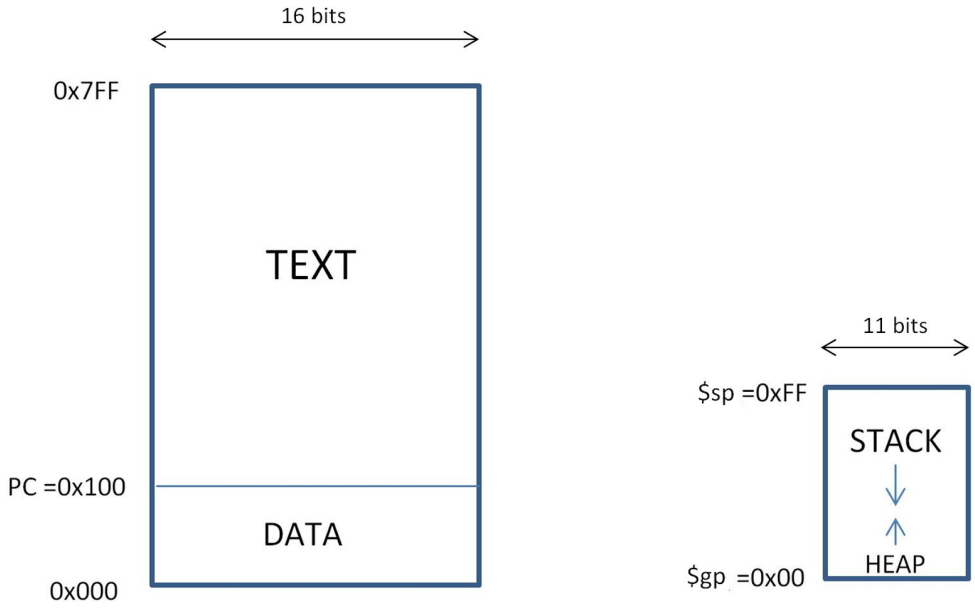
System Calls:

Bits	Action
000	Exit
001	Print String from Data at Address in acc.
010	Load Int from Data at Address in acc. Into the acc
100	Print integer from accumulator
101	Print all registers in dec, and hex
110	Print out the stack from top until null in dec and hex

Memory Diagram

Instr Memory Address is 11 bits.
Stack Address is 8 bits.

Instr Memory is 16 bits wide.
Stack is 11 bits wide.



Design Choices:

- **Good design demands good compromises**- Each instruction takes up 1 line in memory regardless if it's an 8 bit or 16 bit instruction. Each 8 bit instruction wastes 8 bits of memory, but having one instruction per line simplifies the hardware.
- Used variable length opcodes in order to maximize the number of instructions
- Based length of instruction memory by maximum jump size (by number of available bits in the Jal command)
- **Simplicity favors regularity-** All R type instructions are two bits (either 00, 0, orr 10). All others opcodes are 5 bits. If the opcode is 5 bits the two msb are 11.
- **Smaller is faster**- built a relatively small stack in order to maximize usage
- **Make the common case fast -** the $0 register. SInce registers are used so much, always read from them regardless of controls. Since stack memory is slower to pull from, have separate MemRead and MemWrite pins instead of just one WE.
- Built an assembler in order to make the user's life easier- extended this to pseudo instructions in order to simplify the assembly code
- The jrra command is only used as a jump for the $ra8 register, so other inputs are interpreted as system calls, such as load into accumulator or print
- Use an accumulator so instruction can include more bits specifying registers. Allows for more registers than if no accumulator.
- No multiply command, but can multiply with add commands. Included an srl command so can divide by up to (2^8) bits, since division is very hard to implement with subtractions

Assembler:

- The assembler converts basic assembly code into machine code
- Built using python and basic file manipulation- reads a text file line by line and does direct conversion into binary
- Replaces basic variables with their addresses, in order to feed them to the processor
- Converts decimal address for Jal into binary
- Includes errors for too many data items or too many instructions
- Includes a line indicator for the processor, to give it the start of instruction memory
- Assembler 0 fills 1 byte instructions so the machine code has 16 bits per line
- Assembler sees end of file when user types eof
- Manual linking required

Hardware:

See pdf and autocad in Drive
See excel sheet of control logic in Drive

- Program counter is made up of a 3 bit register for the high bits (PC3) and an 8 bit register (PC8) for the low bits of the instruction memory address
- $ra3 and $ra8 follow the same principle. $ra8 is addressable so in hardware $ra3 is tied to $ra8 so $ra3 does what $ra8 does.
- Multiplexers select inputs to various components
- Overall control unit
- One ALU that does add, slt, nand, srl, sub, (and no-op)
- Dedicated PC+1 adder
- Dedicated adder for beq
- For concatenations, the two or three bits concatenated are always the msb
- 8 Addressable registers
- 1 Accumulator
- $acc buffer so can don't have to both read and write to the $acc in one cycle
- $0 register initialized at zero and cannot be written over
- Syscalls if (jrra not $ra8)
- During a printstr syscall loads the lower 8 bits of the starting address of the string into the accumulator. The accumulator is then zero extended to 11 bits. That's how the verilog can grab the correct string. Hence, .data section ends at address 00011111111

<u>Flags</u>

- Flag_V to indicate if ALU result from addition is overflow or underflow. Triggers if try to add two positive numbers and get a negative. Also triggers if try to add two negative values and get a positive.
- Flag_T to indicate if jumps to an unknown address (result of a bad jal, beq, or failure to include an exit syscall). Triggers and terminates program to prevent run-away simulation
- Stack overflow or underflow will never happen since stak address is unsigned and offsets to it are signed, but address can only be 8 bits. So if add say 50 to address 255 the new address will falsely be 49. Stack will be wrong, but will not throw program into oblivion. User can print out the stack periodically to check.

Timing

- Single cycle processor
- 2 ns per cycle
- CPI = 1
- PC reads on positive clock edge
- PC gets written to on negative clock edge
- $acc buffer updates on positive clock edges
- No delays were needed because it is a simulation and everything is instantaneous, however verilog code is built so delays can be inserted at crucial places to see effects (Ex instruction goes through a buffer before going to the rest of the hardware so can delay the instruction, can also delay some multiplexers)

How Program is loaded into Simulation

Verilog is hardcoded to look for machine code in a file named instrmem.list

Paste desired program machine code into instrmem.list

Then recompile and run the verilog

Important note that programs must terminate with the exit syscall (jrra 000). It does not have to be at the end of the program physically, but it must be included for the verilog to end the simulation. (Flags if not found)