BASIC & "KITCHEN SINK" PROCESSOR WRITE-UP

Frank Chiarulli Jr.
Union College

Using the Assemblers:

The assembler's are named BasicAssembler.py and KitchenSinkAssembler.py respectively.

To assemble a bit of code for a given processor, simply write your assembly code into a plain text file (for this example we will be using example.txt and be compiling it for the kitchen sink processor). Then in a terminal simply run the command "python2 KitchenSinkAssembler.py example.txt". This command will assemble the code and save the assembled code as 'filename.hex', so in this example it would output a file named example.hex. You can then load the assembled hex file into the instruction memory of the processor.

Note: The assemblers are *error-checking* so if any errors occur while assembling, a hex file will NOT be written this is a feature not a bug, instead the assembler will print to the console exactly what errors occurred and on what line(s) so that the user can easily find and correct any errors he/she might have made while writing the code. This way every .hex file the assembler outputs will be runnable.

Assembly Code:

The assembly code fits the following notation:

R-type: op $destreg $src1reg $src2reg
I-type: op $destReg $src1reg immediateValue

Note: the assemblers only can handle immediate values in base 10.
Each register address and immediate value for both processors is four bits long.

Both Processors share the following opperations (sudo ops denoted with a *):
Add, addi, sub, subi, or, ori, and, andi, lw, sw, *nop

The Kitchen Sink Processor adds the following operations (sudo ops denoted with a *):
Blt, beq, bne, j, *move, *bgt

Jump notaion: j placetojump
Move notation: move $dest $src

Both assemblers support commenting within the assembly code. Comments are denoted by a # and may be on their own lines, or immediately after the assembly instruction on the instructions line itself.
Both assemblers also support blank lines for easier organization of assembly code.

The Kitchen Sink assembler also adds support for constant values and labels:

Constants should be declared at the top of the assembly code, one per line denoted by this notation:
NAME = 4
Where NAME is the name of the constant and 4 is the value of the constant. Note the constant must be able to be stored in a four bit twos complement number (i.e. -8 to 7 or the assembler will throw an error).

Labels can be made by typing the label name followed immediately by a colon, single space, and then the instruction. The label can then be used elsewhere replacing a jump or branch immediate value.
Ex:

```
loop: addi $1 $0 4
j loop
```

EXAMPLE ASSEMBLY CODE:

An example of a **base processor** program:

```
addi $1 $0 5 # adds 5 to reg 1
#example comment
sw $1 $0 0 #stores 5 into the beginning of memory
```

An example of a **kitchen sink processor** program:

```
IMMEDIATE = 4
j cont
nop
addi $1 $0 IMMEDIATE
cont: addi $2 $0 IMMEDIATE
```

OPPERATIONS EXPANDED

Basic Processor Instructions and corresponding binary op codes:
Basic Processor has 17 bit instruction words.

OPERATION	OP CODE	Next 4 bits	Next 4 bits	Next 4 bits
add	00000	dest reg	src1 reg	src2 reg
addi	10000	dest reg	src1 reg	immediate
sub	00011	dest reg	src1 reg	src2 reg
subi	10011	dest reg	src1 reg	immediate
or	00010	dest reg	src1 reg	src2 reg
ori	10010	dest reg	src1 reg	immediate
and	00001	dest reg	src1 reg	src2 reg
andi	10001	dest reg	src1 reg	immediate
lw	11000	dest reg	base addr reg	immediate offset
sw	10100	src reg	base addr reg	immediate offset

Kitchen Sink Processor Instructions and corresponding binary op codes:
Kitchen Sink Processor has 20 bit instruction words.

OPERATION	OP CODE	Next 4 bits	Next 4 bits	Next 4 bits
add	00000000	dest reg	src1 reg	src2 reg
addi	10000000	dest reg	src1 reg	immediate
sub	00000011	dest reg	src1 reg	src2 reg
subi	10000011	dest reg	src1 reg	immediate
or	00000010	dest reg	src1 reg	src2 reg
ori	10000010	dest reg	src1 reg	immediate
and	00000001	dest reg	src1 reg	src2 reg
andi	10000001	dest reg	src1 reg	immediate
lw	10001000	dest reg	base addr reg	immediate offset
sw	10000100	src reg	base addr reg	immediate offset
beq	00110000	src1 reg	src2 reg	immed. branch val
bne	00100000	src1 reg	src2 reg	immed. branch val
blt	00010000	src1 reg	src2 reg	immed. branch val

Jump works a little bit differently:

OPPERATION	OP CODE	remaining 12 bits
j	01000000	line addr to jump to

*sudo instructions not shown as they are converted into one of the binary operations shown above at assembly time.

DESIGN DECISIONS:

Base Processors Design Decisions:

- Expanded 16 registers, allows for 4 bit addresses so that we can have a useable amount of registers to run relatively complex programs, also allows for more useful immediate values.
- Created an integrated control logic unit to allow for easy expansion of operations without need for unnecessary logic to toggle things such as muxes in the processor itself

Kitchen Sink Design Decisions:

- Hardwired $0 to be immutable.
    - Better you can't write to $0 and keep your constant zero safe than being able to break every instruction relying on $0 being holy after the fact.
- Branching
    - Added =,<,> to the ALU for use with branching
    - Originally implemented beq, blt, and bgt on a hardware level, however switched to bgt to bne because this allows us to branch if not equal, while bgt can be easily implemented into the assembler as a sudo instruction, calling blt and just swapping the two register addresses.
    - Used the existing register 2 read pin to toggle between reading the destination input and the second source input to calculate branch operations.
        - This minimizes the amount of new hardware required for blt, the issue however is because the destination is ported to ALU in B, we actually need to check the GREATER THAN value of the ALU to see if we want to branch because A and B are flipped.
- Jumping
    - Changed instruction memory to be indexed by 12 bits as this is the remaining bits left after the op code.
    - This allows for not only substantially more space for instructions, but also you can just use the remaining 12 non-opcode bits for the jump destination when jumping.
- Labels
    - Labels are denoted by "label: instruction" so that they can easily be parsed out by splitting on ":"
- Comments
    - Denoted by "#" for easy parsing.
- Constants
    - Used "=" instead of standard MIPs notation so that we can easily skip constant lines when parsing the instructions by skipping lines which contain "=" symbols.
    - First removes comments before gathering constant values so that having comments with = symbols in them doesn't break assembling.

- Over-all Assemblers
    - For easy testing and code writing, the assembler will not write a .hex file unless the assembler has valid assembly code that can be ran, that way we know when a .hex file is written that it can be run on logisim with no errors.
    - Because debugging assembly is hard:
        - Have the assembler do it for you. Adding error handling inside of the assembler made life way easier when testing the CPU itself as the assembler would not only let me know of mistakes but also what the mistake was and what line it occurred at.

TEST PROGRAMS

In the project folder you will find two 20+ line programs, BaseTest.txt/BaseTest.hex and KitchenSinkTest.txt/KitchenSinkTest.hex, both which run on their respective processors. The base test can be run on the kitchen sink processor, however it must first be re-compiled using the KitchenSinkAssembler.py.

Note: Before images not included as they are simply all zeros.

Base Test:

The base test tests the various base functions, at the end of the run, the registers and data memory will look as follows:

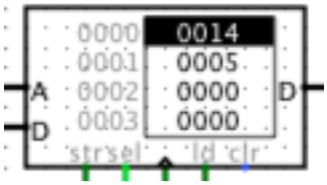
Contents of registers:

$1 = 4, $2 = 1, $3 = 0, $4 = 20(hex 14), $5 = 5, $6 = 5

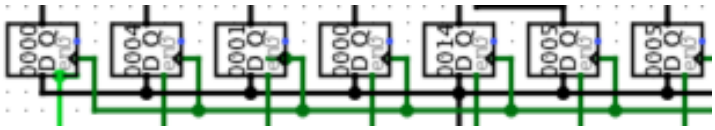
Contents of memory:

0000: 20(hex 14), 0001: 5

Data Memory after test run:



Registers after test run:



Kitchen Sink Test:

The kitchen sink test should have the same outcome as the base test, however it uses branches, jumps, labels, constants, and moves to accomplish the task.

Contents of registers:

$1 = 4, $2 = 1, $3 = 0, $4 = 20(hex 14), $5 = 5, $6 = 5

Contents of memory:

0000: 20(hex 14), 0001: 5

Data Memory after test run:

Registers after test run: