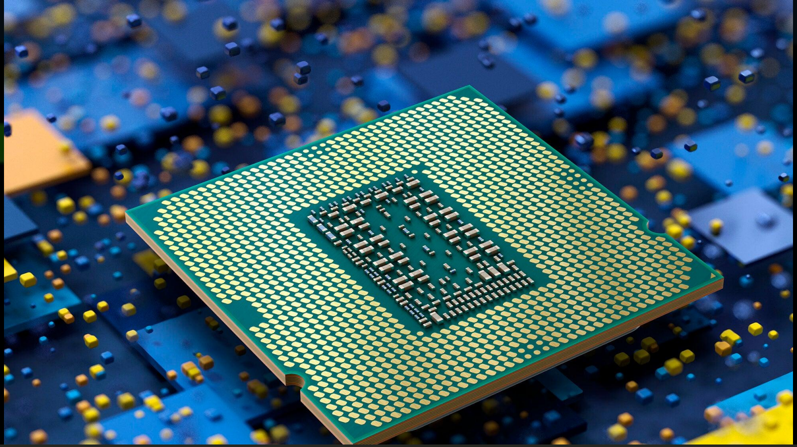
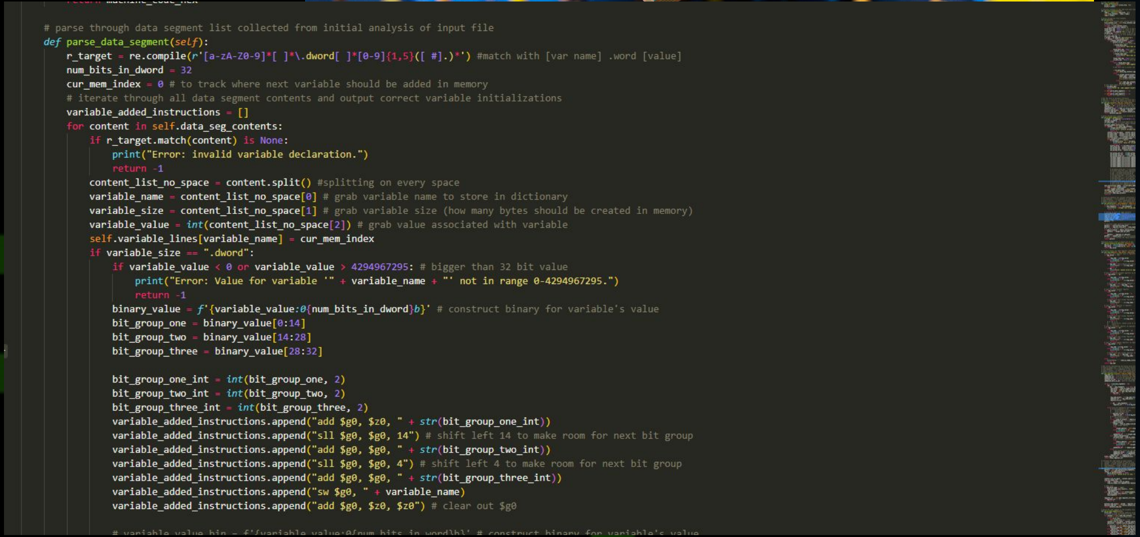
# Processor, Assembly, and Snake





```
# parse through data segment list collected from initial analysis of input file
def parse_data_segment(self):
    r_target = re.compile(r'[a-zA-Z0-9]*[ ]*\.dword[ ]*[0-9]{1,5}([ #].)*') #match with [var name] .word [value]
    num_bits_in_dword = 32
    cur_mem_index = 0 # to track where next variable should be added in memory
    # iterate through all data segment contents and output correct variable initializations
    variable_added_instructions = []
    for content in self.data_seg_contents:
        if r_target.match(content) is None:
            print("Error: invalid variable declaration.")
            return -1
        content_list_no_space = content.split() #splitting on every space
        variable_name = content_list_no_space[0] # grab variable name to store in dictionary
        variable_size = content_list_no_space[1] # grab variable size (how many bytes should be created in memory)
        variable_value = int(content_list_no_space[2]) # grab value associated with variable
        self.variable_lines[variable_name] = cur_mem_index
        if variable_size == ".dword":
            if variable_value < 0 or variable_value > 4294967295: # bigger than 32 bit value
                print("Error: Value for variable '" + variable_name + "' not in range 0-4294967295.")
                return -1
            binary_value = f'{variable_value:0{num_bits_in_dword}b}' # construct binary for variable's value
            bit_group_one = binary_value[0:14]
            bit_group_two = binary_value[14:28]
            bit_group_three = binary_value[28:32]

            bit_group_one_int = int(bit_group_one, 2)
            bit_group_two_int = int(bit_group_two, 2)
            bit_group_three_int = int(bit_group_three, 2)
            variable_added_instructions.append("add $g0, $z0, " + str(bit_group_one_int))
            variable_added_instructions.append("sll $g0, $g0, 14") # shift left 14 to make room for next bit group
            variable_added_instructions.append("add $g0, $g0, " + str(bit_group_two_int))
            variable_added_instructions.append("sll $g0, $g0, 4") # shift left 4 to make room for next bit group
            variable_added_instructions.append("add $g0, $g0, " + str(bit_group_three_int))
            variable_added_instructions.append("sw $g0, " + variable_name)
            variable_added_instructions.append("add $g0, $z0, $z0") # clear out $g0
```

By: Ryan 🤩, Garrett 🐐, Minh 🥶, Matthew 🥵

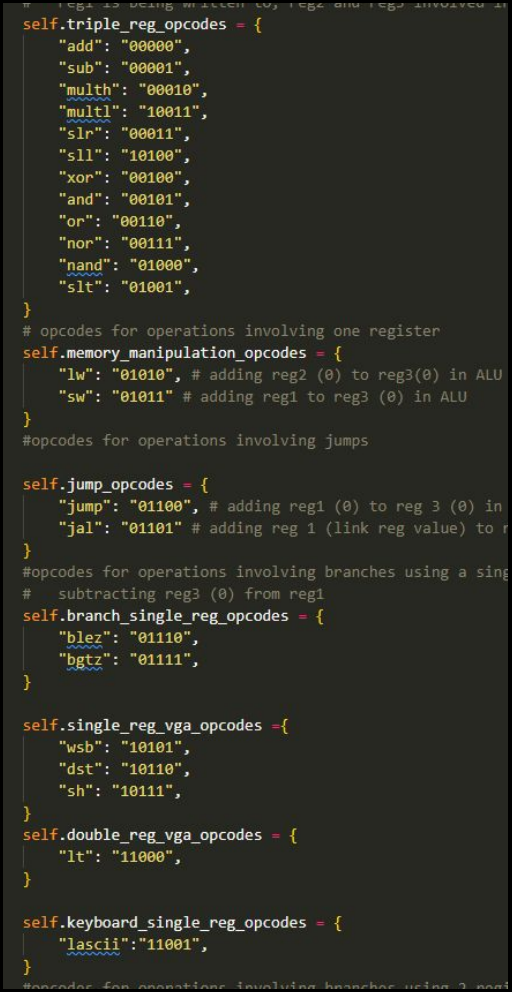
Best: 293

## Our Original Goal

Create a general purpose processor that is different from MIPS so we can program a game on it using our own assembly language

## Assembly language

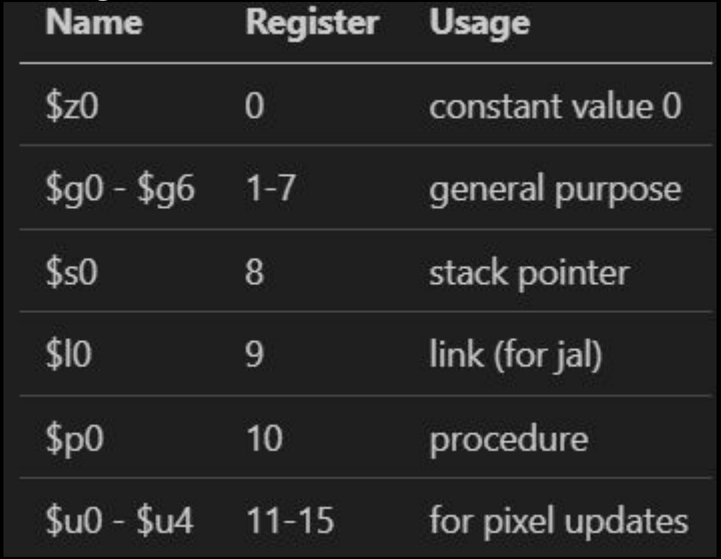
- One type of instruction
- Supports variable declaration (memory interaction)
- Supports jump, jump and link, branches
- Specialty instructions for handling keyboard and VGA



```
#   reg1 is being written to, reg2 and reg3 involved in
self.triple_reg_opcodes = {
    "add": "00000",
    "sub": "00001",
    "multh": "00010",
    "multl": "10011",
    "slr": "00011",
    "sll": "10100",
    "xor": "00100",
    "and": "00101",
    "or": "00110",
    "nor": "00111",
    "nand": "01000",
    "slt": "01001",
}
# opcodes for operations involving one register
self.memory_manipulation_opcodes = {
    "lw": "01010", # adding reg2 (0) to reg3(0) in ALU
    "sw": "01011" # adding reg1 to reg3 (0) in ALU
}
#opcodes for operations involving jumps

self.jump_opcodes = {
    "jump": "01100", # adding reg1 (0) to reg 3 (0) in
    "jal": "01101" # adding reg 1 (link reg value) to r
}
#opcodes for operations involving branches using a sing
#   subtracting reg3 (0) from reg1
self.branch_single_reg_opcodes = {
    "blez": "01110",
    "bgtz": "01111",
}

self.single_reg_vga_opcodes ={
    "wsb": "10101",
    "dst": "10110",
    "sh": "10111",
}
self.double_reg_vga_opcodes = {
    "lt": "11000",
}

self.keyboard_single_reg_opcodes = {
    "lascii":"11001",
}
#opcodes for operations involving branches using 2 regi
```

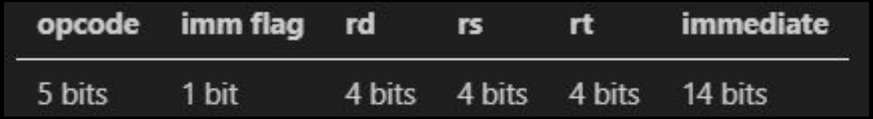
# Registers and Instruction Format

Register Architecture:



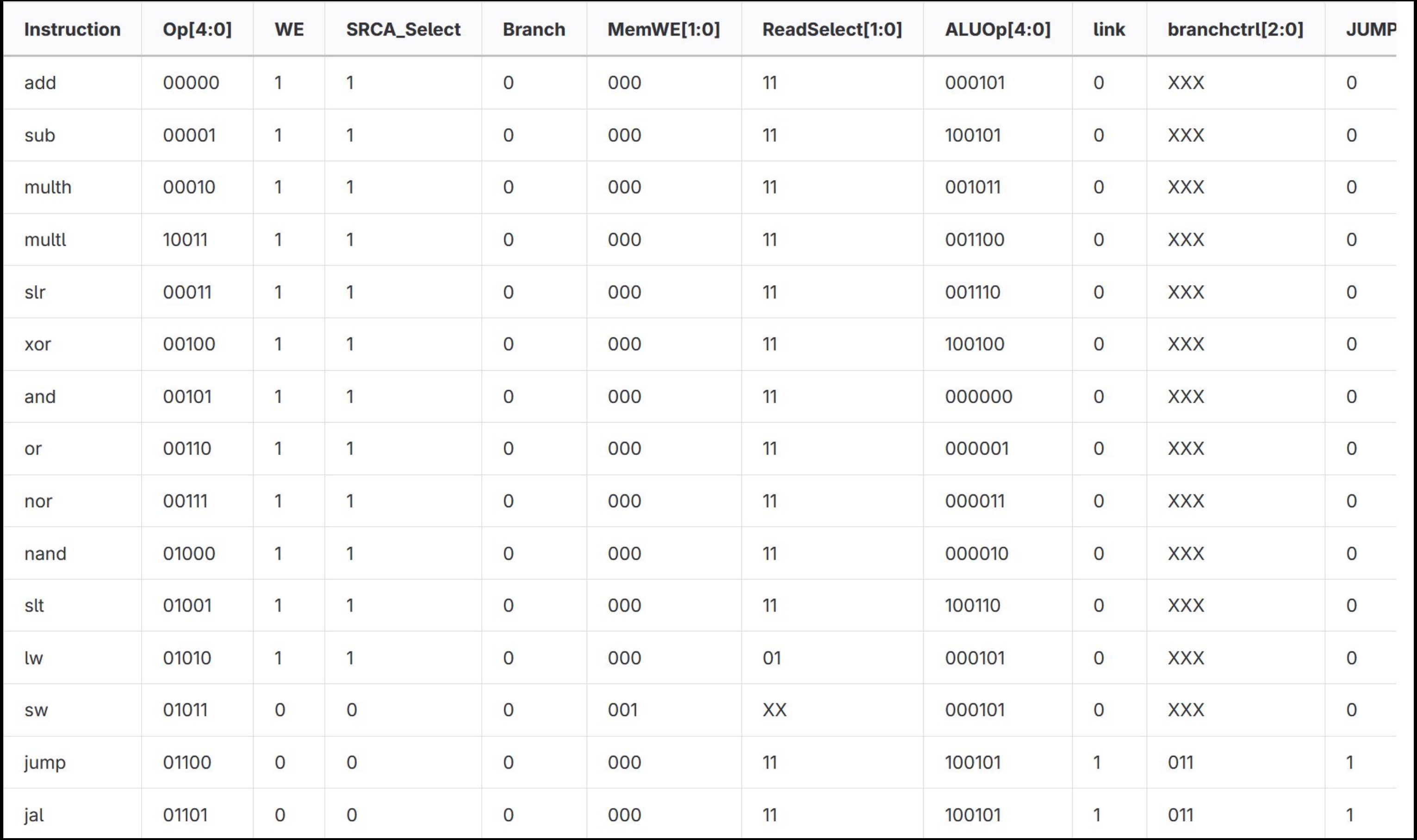
| Name | Register | Usage |
|---|---|---|
| $z0 | 0 | constant value 0 |
| $g0 - $g6 | 1-7 | general purpose |
| $s0 | 8 | stack pointer |
| $l0 | 9 | link (for jal) |
| $p0 | 10 | procedure |
| $u0 - $u4 | 11-15 | for pixel updates |

Instruction Format:



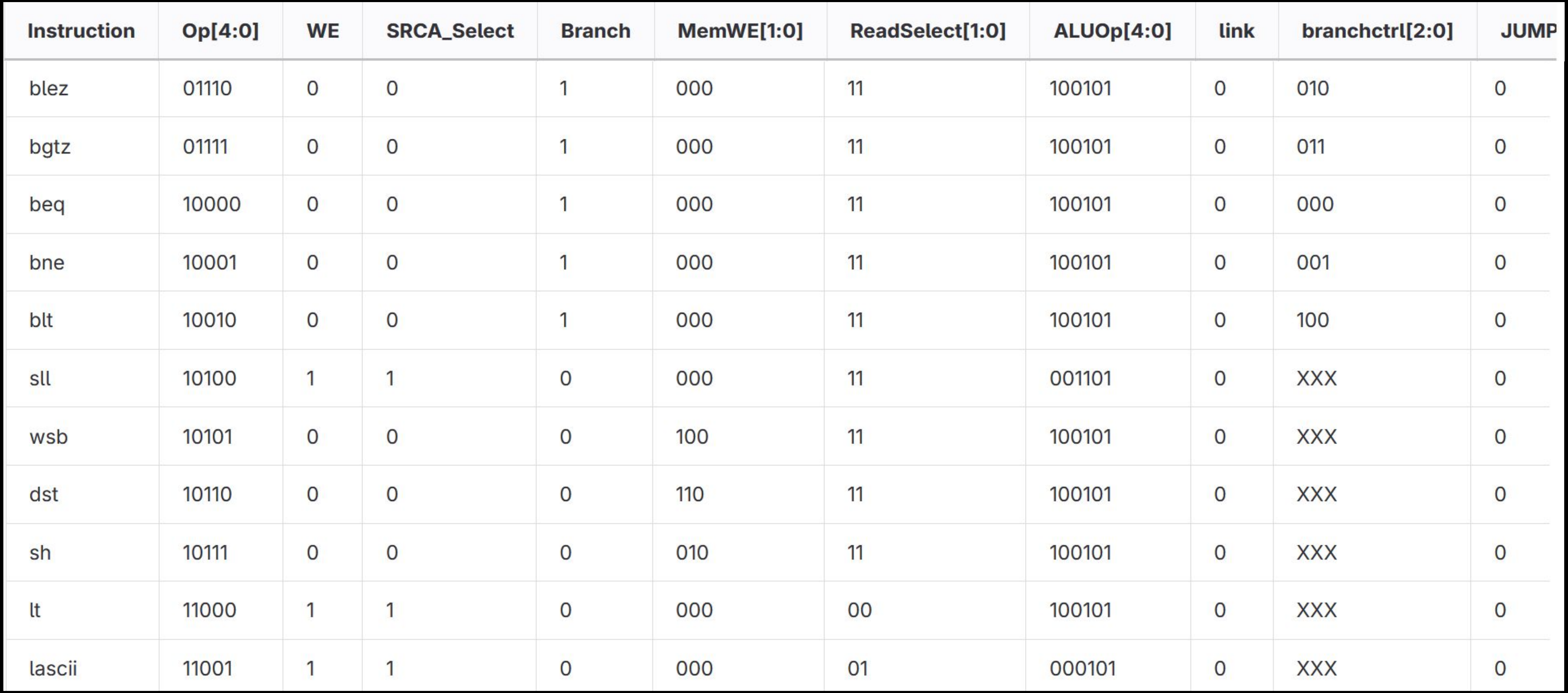
| opcode | imm flag | rd | rs | rt | immediate |
|---|---|---|---|---|---|
| 5 bits | 1 bit | 4 bits | 4 bits | 4 bits | 14 bits |

# Instruction Decoding



| Instruction | Op[4:0] | WE | SRCA_Select | Branch | MemWE[1:0] | ReadSelect[1:0] | ALUOp[4:0] | link | branchctrl[2:0] | JUMP |
|---|---|---|---|---|---|---|---|---|---|---|
| add | 00000 | 1 | 1 | 0 | 000 | 11 | 000101 | 0 | XXX | 0 |
| sub | 00001 | 1 | 1 | 0 | 000 | 11 | 100101 | 0 | XXX | 0 |
| multh | 00010 | 1 | 1 | 0 | 000 | 11 | 001011 | 0 | XXX | 0 |
| multl | 10011 | 1 | 1 | 0 | 000 | 11 | 001100 | 0 | XXX | 0 |
| slr | 00011 | 1 | 1 | 0 | 000 | 11 | 001110 | 0 | XXX | 0 |
| xor | 00100 | 1 | 1 | 0 | 000 | 11 | 100100 | 0 | XXX | 0 |
| and | 00101 | 1 | 1 | 0 | 000 | 11 | 000000 | 0 | XXX | 0 |
| or | 00110 | 1 | 1 | 0 | 000 | 11 | 000001 | 0 | XXX | 0 |
| nor | 00111 | 1 | 1 | 0 | 000 | 11 | 000011 | 0 | XXX | 0 |
| nand | 01000 | 1 | 1 | 0 | 000 | 11 | 000010 | 0 | XXX | 0 |
| slt | 01001 | 1 | 1 | 0 | 000 | 11 | 100110 | 0 | XXX | 0 |
| lw | 01010 | 1 | 1 | 0 | 000 | 01 | 000101 | 0 | XXX | 0 |
| sw | 01011 | 0 | 0 | 0 | 001 | XX | 000101 | 0 | XXX | 0 |
| jump | 01100 | 0 | 0 | 0 | 000 | 11 | 100101 | 1 | 011 | 1 |
| jal | 01101 | 0 | 0 | 0 | 000 | 11 | 100101 | 1 | 011 | 1 |

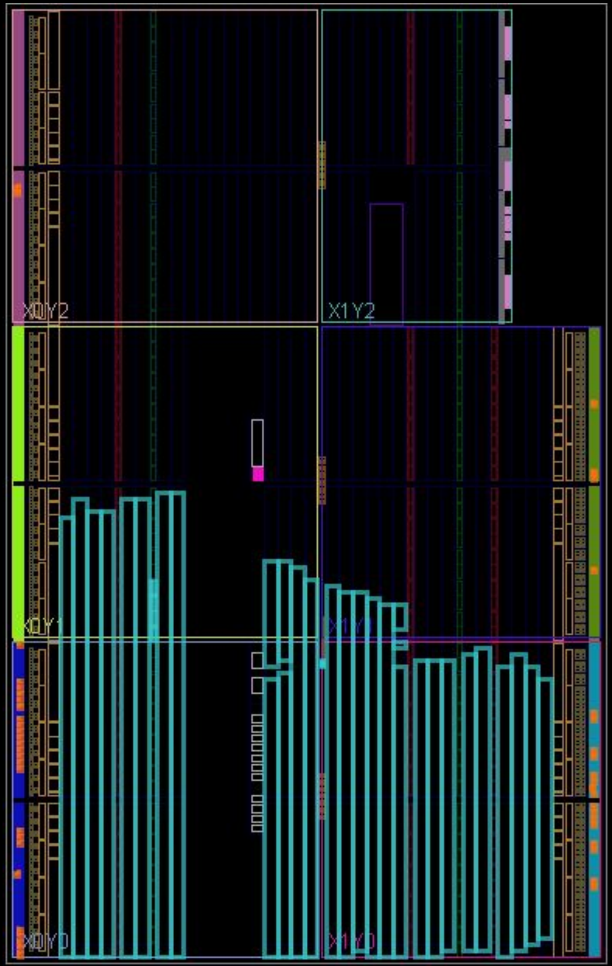
# Instruction Decoding



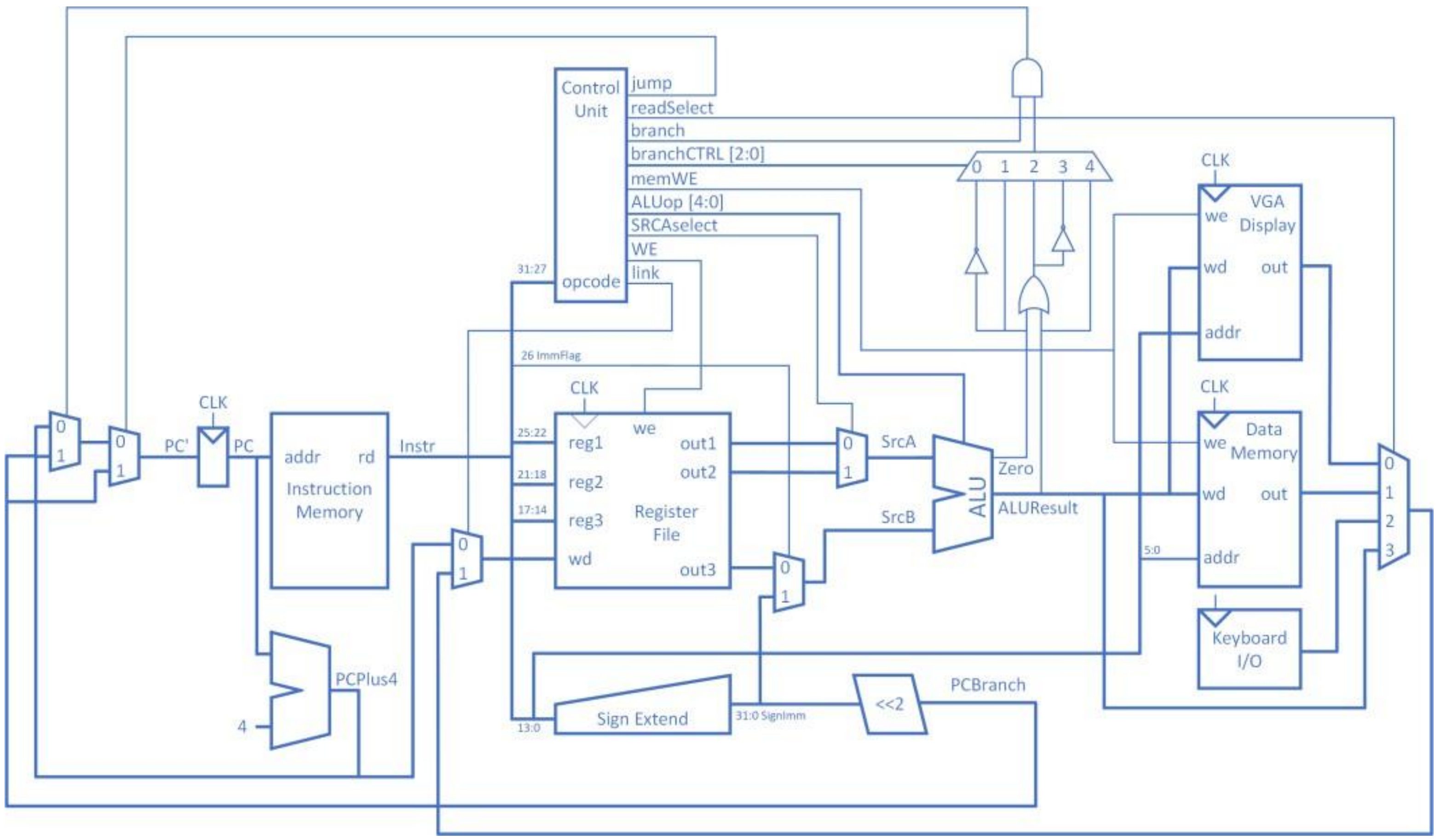
| Instruction | Op[4:0] | WE | SRCA_Select | Branch | MemWE[1:0] | ReadSelect[1:0] | ALUOp[4:0] | link | branchctrl[2:0] | JUMP |
|---|---|---|---|---|---|---|---|---|---|---|
| blez | 01110 | 0 | 0 | 1 | 000 | 11 | 100101 | 0 | 010 | 0 |
| bgtz | 01111 | 0 | 0 | 1 | 000 | 11 | 100101 | 0 | 011 | 0 |
| beq | 10000 | 0 | 0 | 1 | 000 | 11 | 100101 | 0 | 000 | 0 |
| bne | 10001 | 0 | 0 | 1 | 000 | 11 | 100101 | 0 | 001 | 0 |
| blt | 10010 | 0 | 0 | 1 | 000 | 11 | 100101 | 0 | 100 | 0 |
| sll | 10100 | 1 | 1 | 0 | 000 | 11 | 001101 | 0 | XXX | 0 |
| wsb | 10101 | 0 | 0 | 0 | 100 | 11 | 100101 | 0 | XXX | 0 |
| dst | 10110 | 0 | 0 | 0 | 110 | 11 | 100101 | 0 | XXX | 0 |
| sh | 10111 | 0 | 0 | 0 | 010 | 11 | 100101 | 0 | XXX | 0 |
| lt | 11000 | 1 | 1 | 0 | 000 | 00 | 100101 | 0 | XXX | 0 |
| lascii | 11001 | 1 | 1 | 0 | 000 | 01 | 000101 | 0 | XXX | 0 |

## Processor Design

- General purpose 32-bit processor
- Single cycle
- Data memory
- Instruction memory
- VGA memory
- VGA output
- 7-segment output
- Keyboard ASCII input



# Processor Diagram

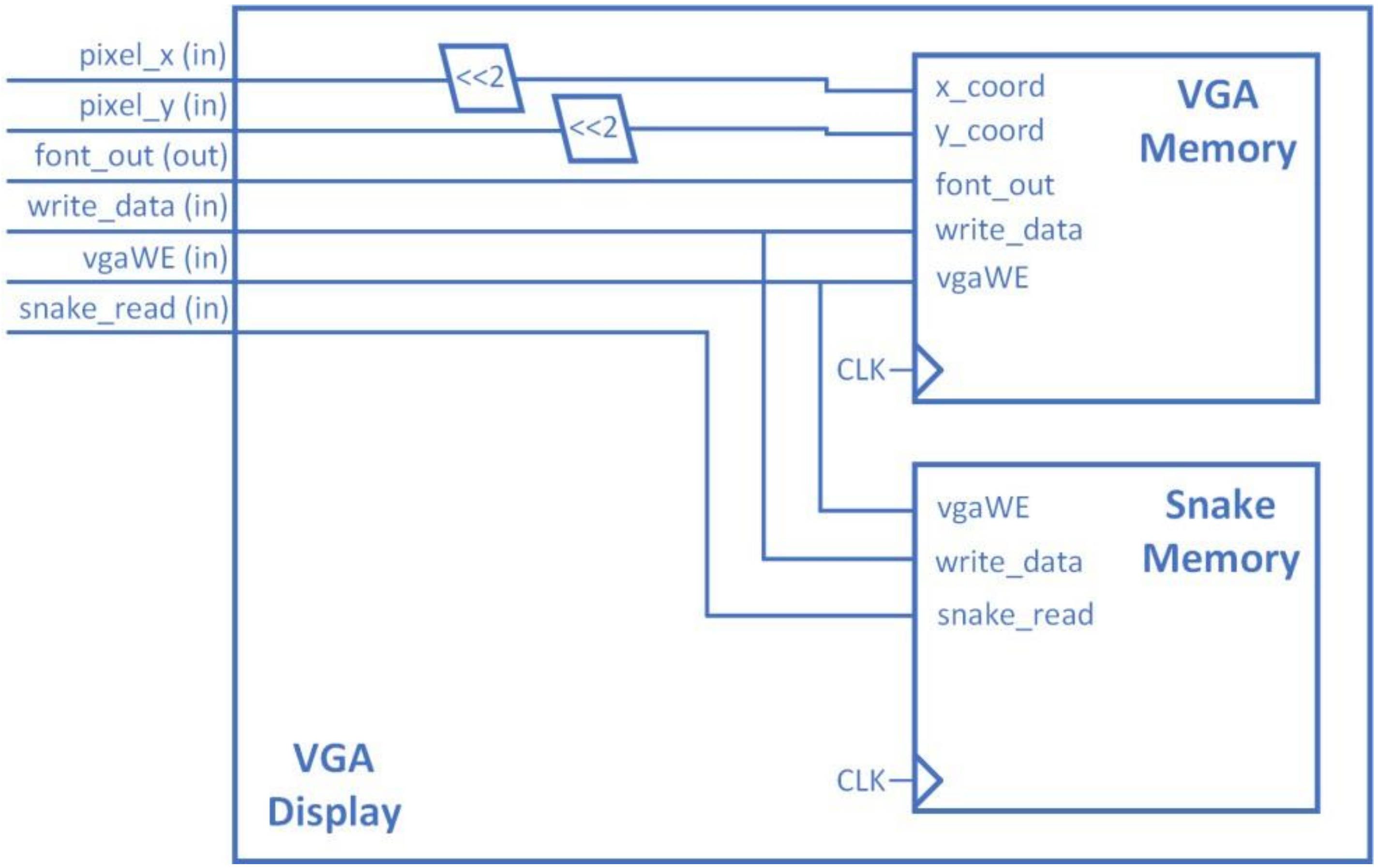


## I/O Design - Display

- VGA sync unit
- VGA memory
- Font gen unit

- Character data stored in array in VGA mem
- Pixel data stored in ROM in font gen unit

# VGA Diagram


pixel_x (in)
pixel_y (in)
font_out (out)
write_data (in)
vgaWE (in)
snake_read (in)
<<2
<<2
x_coord
y_coord
font_out
write_data
vgaWE
VGA Memory
CLK
vgaWE
write_data
snake_read
Snake Memory
CLK
VGA Display

## I/O Design - Keyboard

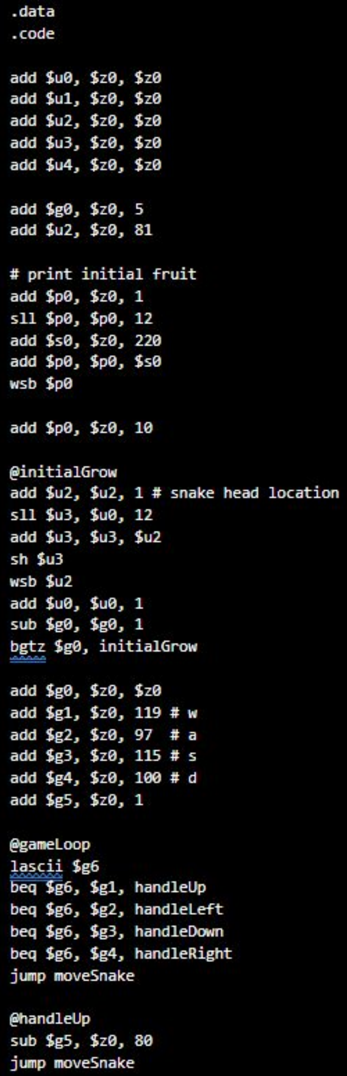
- Receive PS2 data and clock from keyboard
- Translate key value to ascii value, and send to processor
- Use ASCII value to control snake heading (WASD)

## Snake Game

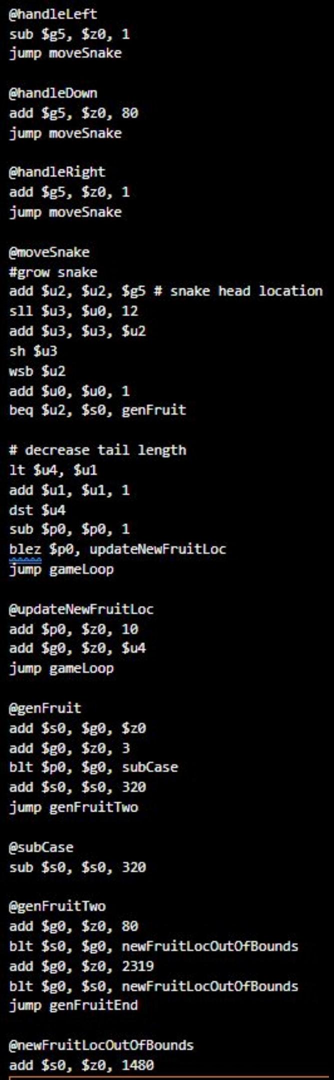
- Classic snake
- Snake grows when eating food
- Snake dies if it collides with wall or its own body

- Food generates at tail position 10 ticks prior
- Snake body is stored in an array in VGA memory

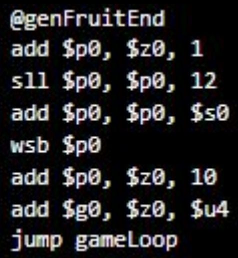
# Snake Game



```
.data
.code

add $u0, $z0, $z0
add $u1, $z0, $z0
add $u2, $z0, $z0
add $u3, $z0, $z0
add $u4, $z0, $z0

add $g0, $z0, 5
add $u2, $z0, 81

# print initial fruit
add $p0, $z0, 1
sll $p0, $p0, 12
add $s0, $z0, 220
add $p0, $p0, $s0
wsb $p0

add $p0, $z0, 10

@initialGrow
add $u2, $u2, 1 # snake head location
sll $u3, $u0, 12
add $u3, $u3, $u2
sh $u3
wsb $u2
add $u0, $u0, 1
sub $g0, $g0, 1
bgtz $g0, initialGrow

add $g0, $z0, $z0
add $g1, $z0, 119 # w
add $g2, $z0, 97  # a
add $g3, $z0, 115 # s
add $g4, $z0, 100 # d
add $g5, $z0, 1

@gameLoop
lascii $g6
beq $g6, $g1, handleUp
beq $g6, $g2, handleLeft
beq $g6, $g3, handleDown
beq $g6, $g4, handleRight
jump moveSnake

@handleUp
sub $g5, $z0, 80
jump moveSnake
```



```
@handleLeft
sub $g5, $z0, 1
jump moveSnake

@handleDown
add $g5, $z0, 80
jump moveSnake

@handleRight
add $g5, $z0, 1
jump moveSnake

@moveSnake
#grow snake
add $u2, $u2, $g5 # snake head location
sll $u3, $u0, 12
add $u3, $u3, $u2
sh $u3
wsb $u2
add $u0, $u0, 1
beq $u2, $s0, genFruit

# decrease tail length
lt $u4, $u1
add $u1, $u1, 1
dst $u4
sub $p0, $p0, 1
blez $p0, updateNewFruitLoc
jump gameLoop

@updateNewFruitLoc
add $p0, $z0, 10
add $g0, $z0, $u4
jump gameLoop

@genFruit
add $s0, $g0, $z0
add $g0, $z0, 3
blt $p0, $g0, subCase
add $s0, $s0, 320
jump genFruitTwo

@subCase
sub $s0, $s0, 320

@genFruitTwo
add $g0, $z0, 80
blt $s0, $g0, newFruitLocOutOfBounds
add $g0, $z0, 2319
blt $g0, $s0, newFruitLocOutOfBounds
jump genFruitEnd

@newFruitLocOutOfBounds
add $s0, $z0, 1480
```



```
@genFruitEnd
add $p0, $z0, 1
sll $p0, $p0, 12
add $p0, $p0, $s0
wsb $p0
add $p0, $z0, 10
add $g0, $z0, $u4
jump gameLoop
```

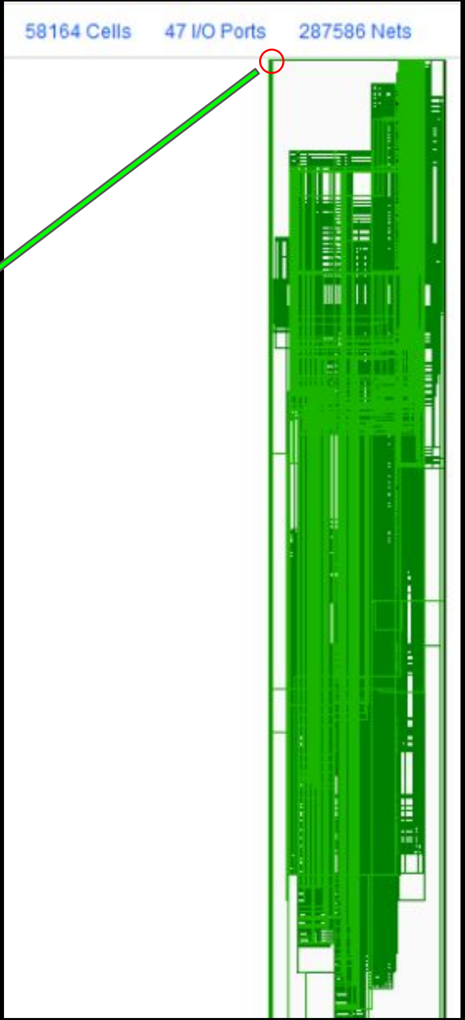
## Challenges faced

- Making working assembler
- Adding new components
- Managing memory sizes
- Black box and signal size issues
- Clock signal mismatching
- Determining control signals

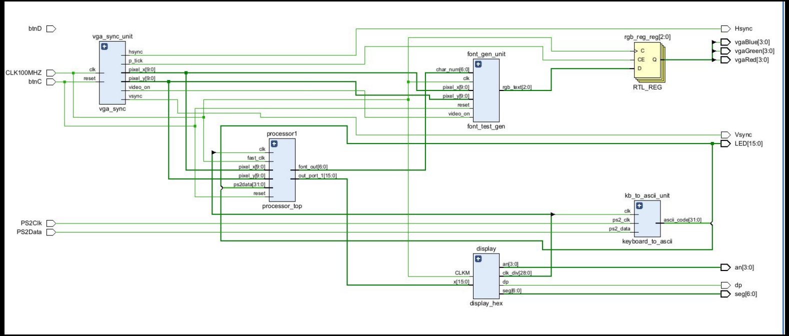
## What we learned

- VHDL is difficult
- Finding small errors is hugely difficult
- Be open to changing the design (don’t fall in love with your ideas)
- Test iteratively
- The hardware diagram is very helpful for planning and fixing errors
- Be careful of comparators

## A cautionary tale…



This is our VGA memory



This is the processor

## Thank you to our alumni!🥰

For allowing us to have the BASYS 3 boards