VLSI System Design

Vincent Immler
vincent.immler@oregonstate.edu

Prepared for: ECE474/574, Spring 2024.

Oregon State University

rootoftrust.io

Microcontroller Design

- **First Step:** Decide on your architecture
  - Stack-based
  - Single working register (accumulator)
  - Multiple general purpose register
  - …
- **Decision depends on your needs!**
  - Need high speed: multiple GPR and instructions
  - Need smallest area: go for stack based (zero registers!)
  - Something in between: go for accumulator based (a compromise between the other two)
  - All require memory!
  - Van-Neumann vs. Harvard?
- **Register count:**
  - Number of registers determines operational efficiency
  - Memory access is costly (minimum a cycle spent; in reality: much worse)

Accumulator-based Architecture

We want to add contents of two RAM locations and write the result into a third location, e.g.:

```
mem[c] <- mem[a] + mem[b]

// only three lines of for accumulator based architecture

LDA A     :     ACC <- mem[A]
ADD B     :     ACC <- ACC + mem[B]
MVA A     :     mem[c] <- ACC
```

Stack-based Architecture

We want to add contents of two RAM locations and write the result into a third location, e.g.:

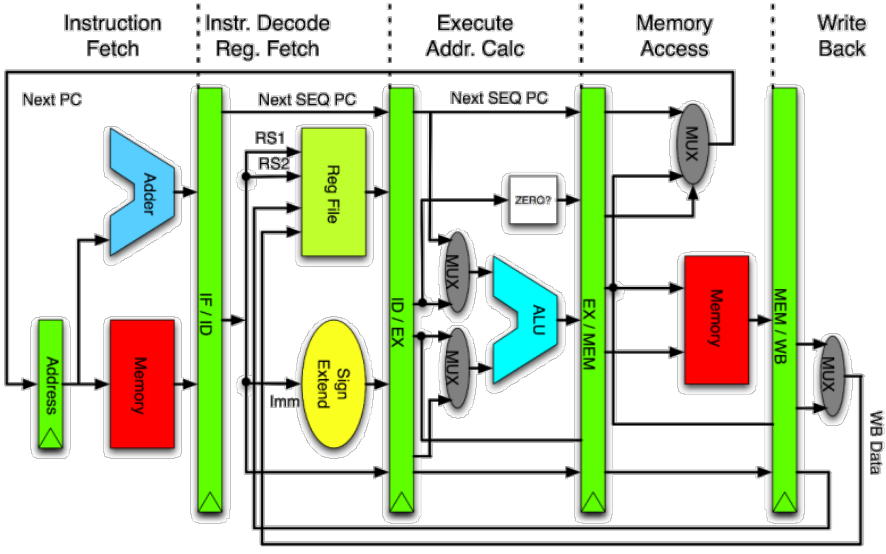
```
mem[c] <- mem[a] + mem[b]

// seven lines of for accumulator based architecture

PTR A     :     PTR <- A
POP       :     mem[T] <- mem[A]
PTR B     :     PTR <- B
POP       :     mem[T+1] <- mem[B]
PTR C     :     PTR <- C
ADD       :     mem[T] <- mem[T] + mem[T+1]
PUSH      :     mem[C] <- mem[T]
```

Requires a dual-port memory!

Stages of Executing Code (Example Architecture)



Stages of Executing Code (Notes)

Typical 5-stage instruction cycle (popular interview question)

- Fetch
- Decode
- Execute
- Memory Access
- (Data/Register) Write-Back

Basic computer organization and what we need to focus on:

- Instruction set architecture: computer architect's view (e.g., ARM)
- Machine organization: logic designer's view
  - Map instructions to logic gates and registers
  - Have them working together in harmony
- Physical implementation: chip designer's view (luckily, we have FPGAs)

Microcontroller Considerations Part 1

- GPR based architecture is advantageous when several temp values and/or constant have to be processed over and over again without need to access them from memory
- The choice of a certain architecture is also influenced by the amount of program memory available, as well as its structure
- If we are given only a limited program memory and the application requires fairly complex operations, go for GPR
- If program memory is relatively cheap and operations are simple, go for ACC
- Most of the time, we need a microcontroller in place of a complex state machine in our overall system
- Furthermore, we do not have the luxury of custom parsers, compilers, etc.
- Go with simplest possible architecture with the simplest possible coding

Microcontroller Considerations Part 2

- Let's assume we checked system requirements and go with GPR based architecture
- The next step is the selection of the instruction set
- Conventional wisdom suggests, we should at least have basic logic operations
  - NAND, NOR, XOR — and ADD
- Furthermore, shift/rotate operations are always useful to implement simple multiplication, division, etc.
- Jump instructions are a must, not only for loops, but also subroutines
- Memory access (read/write) instructions are also needed
- Of all these instructions, we are missing the ability to assign fixed numbers (constants) to program code, this is called the load immediate instruction
- Let's now compile a list of all instructions and assign an opcode to each of them

Instruction	Opcode	Description	
NAND	0000	Logical NAND	$Rd \leftarrow Rs1$ OPR $Rs2$
NOR	0001	Logical NOR	
XOR	0010	Logical XOR	
ADD	0011	Arithmetic ADD	
ROTL	0100	Rotate left	$Rd \leftarrow Rs1 \lll Rs2[n-1:0]$
SLA	0101	Shift left arithmetic	$Rd \leftarrow Rs1 \ll Rs2[n-1:0]$
SRA	0110	Shift right arithmetic	$Rd \leftarrow Rs1 \gg Rs2[n-1:0]$
SRL	0111	Shift right logical	
JMPC	1000	Jump conditional	$PC \leftarrow Rd$ if $(Rs1 == 0)$ else $PC+1$
JMPD	1001	Jump direct	$PC \leftarrow$ #Adr
JMPS	1010	Jump to subroutine	$PC \leftarrow Rd$, Stack $\xleftarrow{push}$ $PC+1$
RETS	1011	Return from subroutine	$PC \xleftarrow{pop}$ Stack
MEMR	1100	Memory read	$Rd \leftarrow mem[Rs1]$
MEMW	1101	Memory write	$mem[Rs1] \leftarrow Rd$
LDI	1110	Load immediate	$Rd \leftarrow$ #Imm
RSVD	1111	Reserved	

Instructions: Initial Considerations

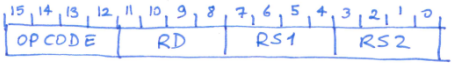
- Length of the opcode depends on the number of instructions
- In this case, we have < 16 instructions, therefore, a 4-bit opcode is sufficient
- Another important parameter is the register width and the number of registers
- Register width is again a generic system parameter
- If data processed in the system is 8-bits, register size should be 8 bits
- On the other hand, # of registers depends on the temporary variables needed in the code
- By writing pseudo code for the system, we can determine the total number of registers we need to make memory access efficient (i.e., minimize the number of accesses)
- Let's assume we decide to go for 16 general purpose registers of 8 bit
- This also determines the data memory width to be 8 width
- Since our instructions require three operands, i.e., three register addresses, we need another 12-bit for this in addition to the 4-bit opcode $\longrightarrow$ 16 bit in total

Instructions: Investigating Details

- Shift/rotate instructions
- JMPC/JMPS
- Jump direct
- JMPS/RETS
- Memory read/write
- Load immediate
- Reserved

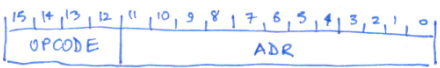
Discussion / work together on whiteboard

The resultant instruction formats:

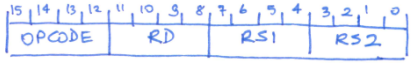


Valid for NAND, NOR, XOR, ADD, ROTL, SLA, SRA, SRL instructions

Valid for JMPC, JMPS, RETS



Valid for JMPD - ADR LS 10/12 bits



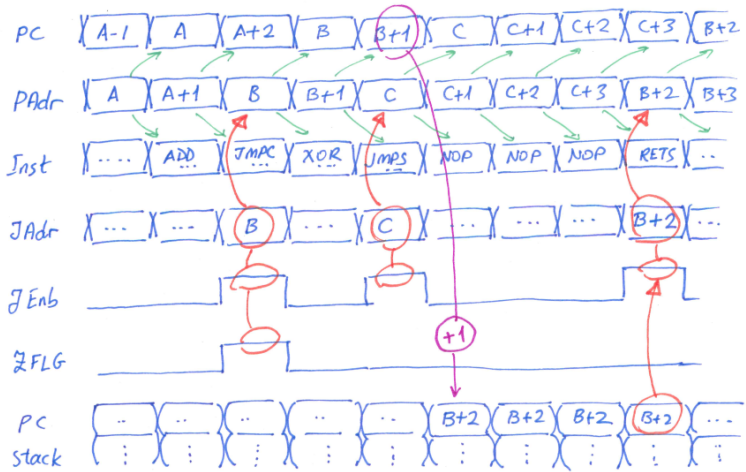
Valid for MEMR, MEMW
Memory address can be either RS1 or a combination of RS1 and RS2 (in case data mem depth > 256)

Valid for LDI

Let's look at a simple code segment and the instruction timing. It should help us to design the program counter, the main and most important part of the design.

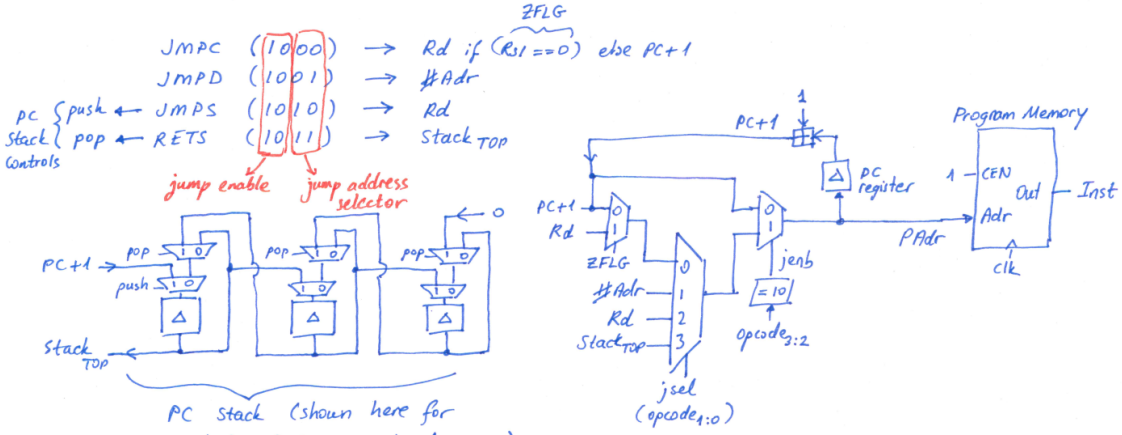
Line NO	Instruction
A	ADD R3 R1 R2
A+1	JMPC R3 R4
⋮	⋮
B	XOR R3 R3 R7
B+1	JMPS R5
⋮	⋮
C	NOP
C+1	NOP
C+2	NOP
C+3	RETS
⋮	

* It is assumed that R4 content is "B", R5 content is "C".



Timing Diagram: Insight

- As seen in the timing diagram, program counter register always gets its next value from program address (PAdr) which is also the input address for the program memory.
- However, PAdr value can be either PC+1 or one of the jump addresses (Rd, #Adr, StackTOP) depending on the existence of a jump instruction (JMPC, JMPD, JMPS, RETS)



Let's now identify the operators for each instruction, which will help us design the register bank and the rest of the micro:

NAND
NOR
XOR
ADD
ROTL
SLA
SRA
SRL

All these instructions have the format

$Rd \leftarrow Rs1\ OP\ Rs2$

Therefore, they require outputs of registers pointed by Rs1 and Rs2. ⇒ Two independent multiplexers at the outputs of the registers, selected by Rs1 and Rs2. Rd is the destination register meaning that only the write enable of the register pointed by Rd (out of 16 regs) shall be "1". Others all "0"!

JMPC
JMPS

They require both outputs of registers pointed by Rd and Rs1. Rs1 is already there due to ALU/shift/rotate instructions. However instead of Rs2, we need to use Rd as register select of the second multiplexer ⇒ Multiplexer select also has to be multiplexed:

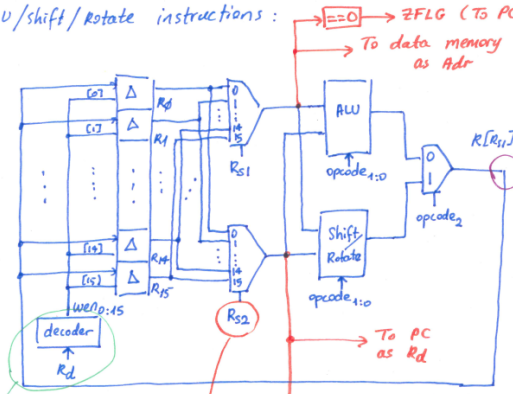
inst == JMPC
OR inst == JMPS
OR inst == MEMW

MEMR → Rd will be the destination register, Rs1 output will be the memory address. No need for special logic.

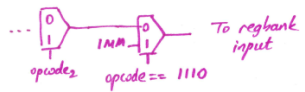
MEMW → Rs1 still memory address. However memory input will come from register pointed by Rd similar to JMPC/JMPS instructions. Add it to OR.

Let's draw the micro architecture for each instruction or instruction group:

– ALU/shift/Rotate instructions:

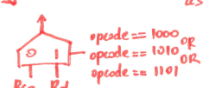


For LDI instruction register input shall be the Imm value changing this part as follows:

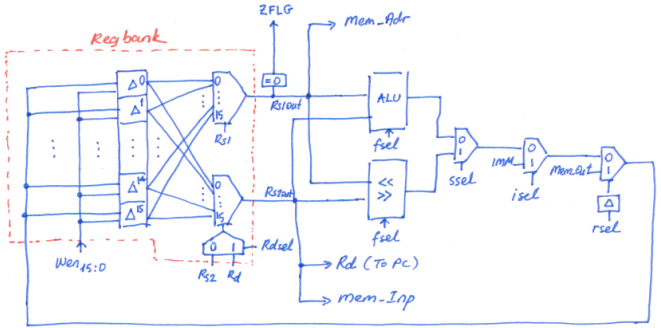


For MEMR, this portion needs further threatment. After data memory is supplied with address from R[RS1], its output becomes available at the next clock and should be written to the register pointed by Rd in the previous clock resulting in the following circuit:

For JMPC/JMPS/MEMW instructions replace this part as follows:



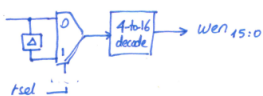
Finally, we put all the things together and obtain a first look at the micro:



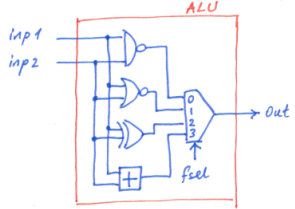
All the select signals should be generated inside the instruction decoder as follows:

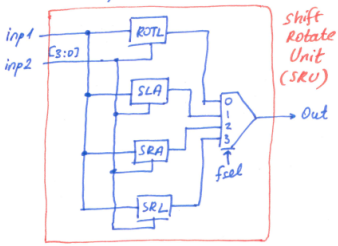
Rs1 = inst[7:4]
Rs2 = inst[3:0]
Rd = inst[11:8]
imm = inst[7:0]
fsel = inst[13:12]
ssel = inst[14]
isel = (inst[15:12] == 1110)

rsel = (inst[15:12] == 1100)
Rdsel = (inst[15:12] == 1000) || (inst[15:12] == 1010)
|| (inst[15:12] == 1101)

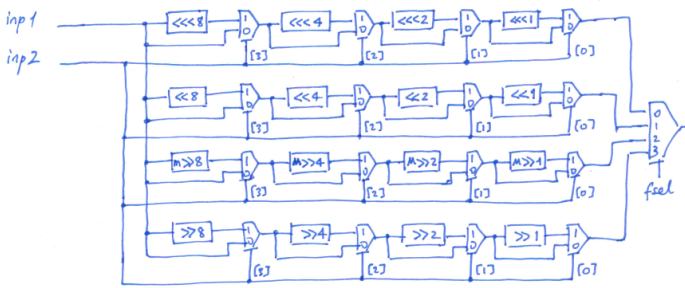


Let's also look at the inside of the ALU and Shift/Rotate Unit :





Inside of SRU :

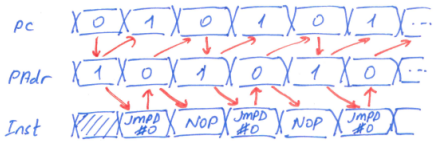


<<<N : Rotate left by N bits

<<N : Shift N "0" bits left.

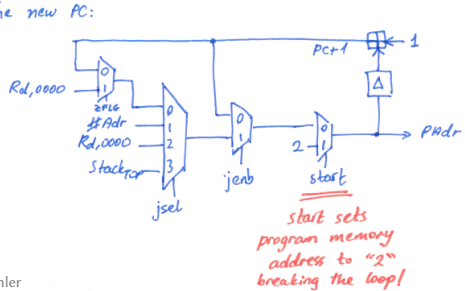
Note that M>>N means MSB bit of input shifted N times while >>N means "0" shifted N times towards right.

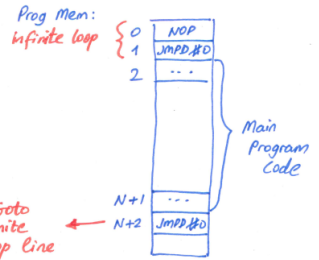
The micro together with the program counter, program memory, data memory will work this way. However there is still one important thing missing: When will the micro start working and when will it stop? Normally for a regular state machine, we have "start" signal that starts its execution and "ready" signal that signals end of operation. In the case of the micro, we will simply have a hybrid (soft + hard) solution. Remember: Upon reset PC ← 0 and PAdr = 1. Simply put NOP to line-0 of program memory and JMPD #0 to line-1 of program memory. Program flow will look like the following:



This way, the program will be locked in this infinite loop.

In order to break the infinite loop, we shall simply use the "start" signal. We will integrate it into the PC in such a way that when start comes PAdr will be "2" making the program jump to line-2 where the actual program code starts. In the end of the execution (i.e. as the last line of code), the program will jump to line-0 putting it into the initial infinite loop until the next "start". Ready can be implemented by writing some flags to a specific location inside the data memory and/or register bank.





Microcontroller Result

Our microcontroller has the following main characteristics:

- Data memory size: 256 bytes
- Program memory size: 4096 words (of 16 bits each) = 8192 bytes (first 2 words reserved)
- Interface with the external user: Start signal and dual port data memory

The next logical question is what can we do with this microcontroller?

- Basic control/procedural operations
- Usually we want to do more than that, such as complex operations
- Example: large modular numbers, finite field operations, etc.
- This can be achieved only via use of a coprocessor
- We can design a coprocessor and integrate it into our microcontroller
- This can be done through the 1111 (reserved) instruction

TOP Module Entity (VHDL)

```
entity TOP_MODULE is
port (
CLK : in std_logic; -- CLOCK
SR : in std_logic; -- RESET
CE : in std_logic; -- CE
START :  in std_logic; -- START
-- synopsys translate_off
dbg_reg1 :  out std_logic_vector(cRegWidth-1 downto 0); -- required for testbench
dbg_reg2 :  out std_logic_vector(cRegWidth-1 downto 0); -- required for testbench
OPRESULT : out std_logic_vector(cRegWidth-1 downto 0); -- result of operation
-- synopsys translate_on
INSTRUCTION :  out std_logic_vector(cInstWidth-1 downto 0)
);
end TOP_MODULE;
```

Please use the same port/entity description in Verilog!

Coprocessor (not relevant for project assignment)

Think of a sequential multiplier with start/ready signals and A/B/M registers, where A and B are inputs and M is the multiplication output. For our example, we assume that A and B are 256 bits wide resulting in a 512 bit M result. Observations:

- Our microcontroller data width is 8 bits. We only have a single instruction for the coprocessor (with possible configuration options). Then we should be able to fill A and B registers via pushing from a source register to A/B registers
- The same for the result. It has to be popped from M into a destination register
- We need a start instruction to generate start/pulse for the sequential multiplier
- Another instruction is necessary to halt the microprocessor until ready

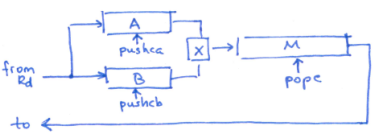
The instructions we need:

- pushca Rd : Push contents of Rd into A register of coprocessor.
- pushcb Rd : Push contents of Rd into B register of coprocessor.
- popc Rd : Pop from M register of coprocessor into Rd.
- startc : Send a "start" pulse to coprocessor.
- waitc : Wait (halt execution) until coprocessor sends "ready" pulse.

Let's also propose possible encodings for these instructions.

	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
pushca	1 1 1 1	Rd	—	0 0 0 0
pushcb	1 1 1 1	Rd	—	0 0 0 1
popc	1 1 1 1	Rd	—	0 0 1 –
startc	1 1 1 1	–		0 1 – –
waitc	1 1 1 1	–		1 – – –

The coprocessor should look like

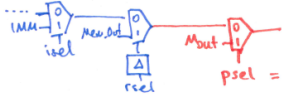


Note that both pushed and popped data starts with LS byte!

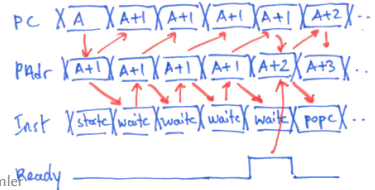
Rd is the source register pointer for pushca and pushcb instructions, which means that we have to modify Rdsel as follows:

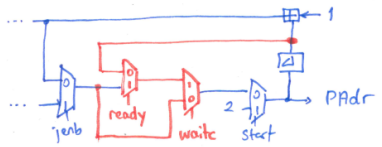
Rdsel = (inst[15:12] == 1000) || (inst[15:12] == 1010) || (inst[15:12] == 1101) ((inst[15:12] == 1111) && (inst[3:1]==001))

We have to define Mout (LS byte) as input to destination register for popc instruction changing the input logic as follows:



Start is pretty straightforward, but waitc/ready requires that PC halts when waitc comes and stays so until ready. PC should be modified as follows:





Putting Everything Together

Once you manage to put everything together, you have a first working microprocessor with coprocessor! Congratulations!

- This is a very basic design only for educational purposes
- However, the principles/ideas are all the same compared to an actual microprocessor
- You gained a lot of experience by completing the previous assignments and projects
- I hope it will serve you well in industry!
- Best wishes and continued success!