Large pipe project 2023

Yosi Ben-Asher
CS, University of Haifa

Abstract

In this project we extend the "normal" 5-stages pipeline RISC-V cpu to a larger pipeline that can include multiple stages of Execute and Mem arranged in different combinations. For example instead of the

$$IF \Longrightarrow D \Longrightarrow E \Longrightarrow M \Longrightarrow W \quad instructions: \ reg1 = (*reg2); \ or \ reg1 = reg2 * reg4;$$

we get a 7-stages pipeline with two memory-stages (connected to a dual-port memory) and two Execute stages that can execute more complex instructions

$$IF \Longrightarrow D \Longrightarrow E \Longrightarrow M \Longrightarrow E \Longrightarrow M \Longrightarrow E \Longrightarrow W \quad instructions: \ reg1 = (*reg2) + (*reg3) * reg4;$$

1 Project description

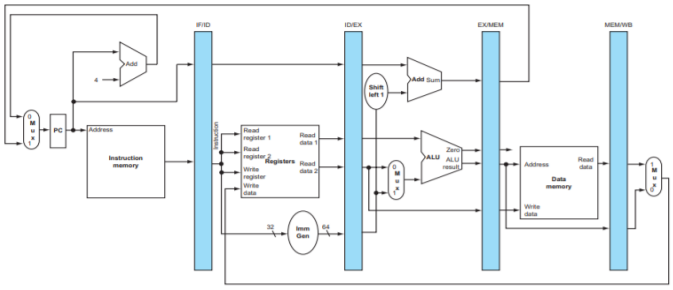

FIGURE 4.33 The pipelined version of the datapath in Figure 4.31. The pipeline registers, in color, separate each pipeline stage. They are labeled by the stages that they separate; for example, the first is labeled *IF/ID* because it separates the instruction fetch and instruction decode stages. The registers must be wide enough to store all the data corresponding to the lines that go through them. For example, the IF/ID register must be 96 bits wide, because it must hold both the 32-bit instruction fetched from memory and the incremented 64-bit PC address. We will expand these registers over the course of this chapter, but for now the other three pipeline registers contain 256, 193, and 128 bits, respectively.

Figure 1: *The 5-stages pipeline of the RISC-V CPU.*

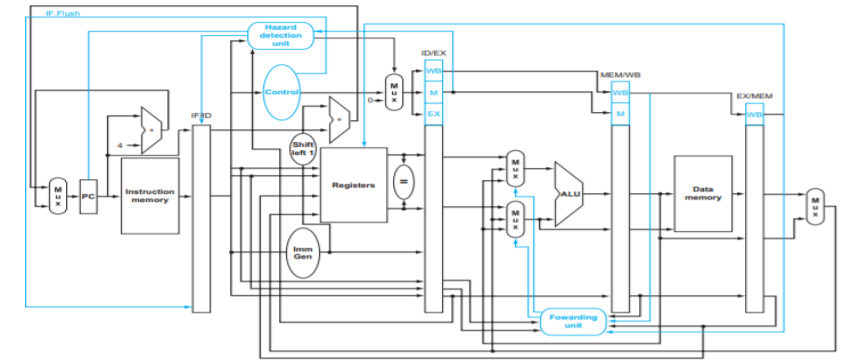

FIGURE 4.62 The final datapath and control for this chapter. Note that this is a stylized figure rather than a detailed datapath, so it's missing the ALUsrc Mux from Figure 4.55 and the multiplexor controls from Figure 4.49.

Figure 2: *Schematics of the 5-stages pipeline RISC-V CPU.*

- Download (green button +zip-download) the 5-stages RISC-V cpu from https://github.com/princeofpython/C Architecture . The pipeline is depicted in figure 1 and the cpu schematics in figure 2. The CPU Verilog files are in the folder "Pipelined CPU" but you may need files from other directories. Read the Verilog code (1102 lines in total) and understand it. Create a a project at Xilinx Vivado, synthesize the RISC-V 5-stages cpu and run simulations using *cpu_tb.v* applied simulation to the executable code of the *.mem files.

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
fm | pred | succ		rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

Figure 3: *RISC-V instructions.*

- The RISCV instructions are given in figure 3. For a regular pipeline as the one used in the above RISC-V cpu) $E \Longrightarrow M$ each of the RISC-V instructions is executed either at the the E-stage (e.g., $ADD\ R31,\ R0,\ R0$) or at the M-stage. Not that adding the offset to a register in Load/Store instructions (such as $LW\ R9,\ 6(R0)$) is done at the decode-stage as reflected (figure 2) by the near register-file ALU that can add a constant to an argument register. Consider the following sequence

```
00000fb3 //ADD R31, R0, R0                ===> E
00201083 //LH R1, 2(R0)                   ===> D+M
00100103 //LB R2, 1(R0)                   ===> D+M
01009093 //SLLI 8-bits from R1 to R1      ===> E
00811113 //SLLI 4-bits from R2 to R2      ===> E
001101b3 //ADD R3, R1, R2                 ===> E
00002203 //LW R4, 0(R0)                   ===> M
02321263 //bne R4, R3                     ===> E     //to instruction 18. Offset = (1
00601283 //LH R5, 6(R0)                   ===> D+M
00500303 //LB R6, 5(R0)                   ===> D+M
00400383 //LB R7, 4(R0)                   ===> D+M
01029293 //SLLI 8-bits                    ===> E     //from R5 to R5
00831313 //SLLI 4-bits from R6 to R6      ===> E
00638433 //ADD R8, R7, R6                 ===> E
```

```
00540433 //ADD R8, R8, R5                ===> E
00702483 //LW R9, 7(R0)                  ===> D+M
00849463 //bne R9, R8 to instruction 18. ===> E     //Offset = (18-16)*4 = 8
fbdff56f //JAL to instr0. Store PC in R10
00100f93 //ADDI R31, R0, 1. FAIL        ===> E
```

- For the extended pipeline such as $Ea \Longrightarrow Ma \Longrightarrow Eb \Longrightarrow Mb \Longrightarrow Ec$ we create more complex long instructions which contains five RISCV instructions such that:
  - Each instruction contains five RISC-V sub instructions that are fetched from the instruction memory by the IF-unit and is decoded as one.

    ```
    [
    ADD Rt0,R1,R1
    NOP
    ADD Rt0,Rt0,R2
    NOP
    ADD R1,Rt0,R3
    ]
    ```

  - For example the instruction corresponding to $R1 = (*(R2+6))+(*(R3+5))$ is specified by

    ```
    [
    NOP              skip over Ea
    LW Rt0, 6(R2)    executed in Ma->
    NOP              skip over Eb
    LW Rt1, 5(R3)    executed in Mb->
    ADD R1, Rt0, Rt1 executed in Ec
    ]
    ```

    where Rt0/Rt1 are temporary values that are passed forward between the pipeline stages.
  - Instructions may include jumps to skip over some stages in the pipe which is realized by inserting NOP instructions between the RISCV sub instructions. For example the instruction for $R1 = (*(R1 + R2 + 6)) * R3 - R4$ is specified as

    ```
    [
    ADD Rt0, R1,   R2 executed  in Ea->
    LW Rt0, 6(Rt0)    executed in Ma->
    MUL Rt0, Rt0,  R3 executed  in Eb->
    NOP               skip over Mb
    SUB R1, Rt0,   R4 executed  in Ec-> W
    ]
    ```

The skip (NOP+NOP...) indicates that the instruction skips several pipeline stages forward (i.e., skip (create a shortcut) to another stage not immediately stage after the current stage). The skip should be implemented at the decode stage such that the time each instruction spends in the pipeline is proportional to the number of stages it is actually using. Following is an instruction that is executed in Ea and immediately skip to the end of the pipe. This instruction should be five time faster than an instruction that use all stages.

```
[
ADD Rt0, R1,   R2 executed  in Ea->W
NOP
NOP
NOP
NOP
]
```

– Each E/M-stage is also a W-stage as it can update a register in the register-file holding a temporary value of the big instruction. This means that the register file should be able to update different registers in parallel (from multiple stages). The instruction sequence should be such that no parallel updates of the same register can occur.

```
[
NOP
LW+0  R0, 6(R2)    executed in Ma->
NOP
LW+0  R1, 5(R3)    executed in Mb->
ADD+0 R2, R0, R1 executed in Ec   //implicit use of Rt0,Rt1
]
```

- In general the RISC-V objectfile looks as following

```
00000fb3 //[ADD R31, R0, R0 ]              ===> Ea->>W       instruction 1
00201083 //[ LH R1, 2(R0)                  ===> Ea->Ma       instruction 2
00100103 //  LB R2, 1(R0)                  ===> Eb->Mb       instruction 2
01009093 //  SLLI 8-bits ] from R1 to R1   ===> Ec->W        instruction 2
00811113 //[ SLLI 4-bits from R2 to R2     ===> Ea->>W       instruction 3
001101b3 //  ADD R3, R1, R2                ===> Ea           instruction 4
00002203 //  LW R4, 0(R0)                  ===> Ma           instruction 4
02321263 //  bne R4, R3  ]                 ===> Eb->>W       instruction 4
00601283 //[ LH R5, 6(R0)                  ===> Ea->Ma       instruction 5
00500303 //  LB R6, 5(R0) ]                ===> Eb->Mb->>W   instruction 5
00400383 //[ LB R7, 4(R0)                  ===> Ea->Ma       instruction 6
01029293 //  SLLI 8-bits  ]                ===> Eb->>Ec->W   instruction 6
00831313 //[ SLLI 4-bits from R6 to R6     ===> Ea->>Eb      instruction 7
00638433 //  ADD R8, R7, R6                ===> Eb->>Ec      instruction 7
00540433 //  ADD R8, R8, R5 ]              ===> EC->W        instruction 7
```

```
00702483 //[ LW R9, 7(R0)
00849463 //  bne R9, R8 to instruction 18. Offset = (18-16)*4 = 8
fbdff56f //  JAL to instr0. Store PC in R10
00100f93 //  ADDI R31, R0, 1.
```

- Your modified 7-stages cpu should include the following modifications:

  1. Produce RISCV executables either using C-code with LLVM/GCC with target-cpu= RISCV
  2. First change the IF to fetch 20bytes each time out of which process the first sub-instruction only.
  3. Create an encoding for NOPs and extend the state-register that is passed between the stages so that it includes $Rt0, Rt1$.
  4. Add the Eb, Mb, Ec missing stages to the CPU architecture.
  5. Extend the decode stage such that instructions with five RISCV sub instructions will pass through the pipeline (see https://llvm.org/docs/RISCVUsage.html).