



Single-Cycle RISC-V

Labor Computerarchitektur

Inhalt

1 Einführung	1
2 Architektur	2
2.1 Analyse	2
3 Control Unit	3
3.1 Umsetzung	3
3.1.1 Extend	4
3.1.2 ALU	4
3.1.3 ALUOp	4
4 Simulation	5
4.1 Analyse	6
5 Einsatz (optional)	7
5.1 EBS3	7
A Single-cycle architecture	8



1 Einführung

Dieses Labor nähert sich der RISC-V -Architektur mit dem Beispiel eines Prozessors, der einen reduzierten Befehlssatz unterstützt und in der Lage ist, ein kleines Assemblerprogramm auszuführen, indem er jeden Befehl in einem einzigen Clock ausführt (single-cycle). Dieser Prozessor namens



HEIRV32_SC kann simuliert oder direkt auf einem FPGA -Chip eingesetzt werden. Die Verbindung zur Aussenwelt kann über Knöpfe und Leds hergestellt werden.

2 Architektur

Abgesehen von der Inkrementierung des Programmzählers funktioniert der Prozessor rein kombinatorisch. Bei jedem Clockschlag wird also eine Anweisung ausgeführt. Die Schaltung ist wie folgt:

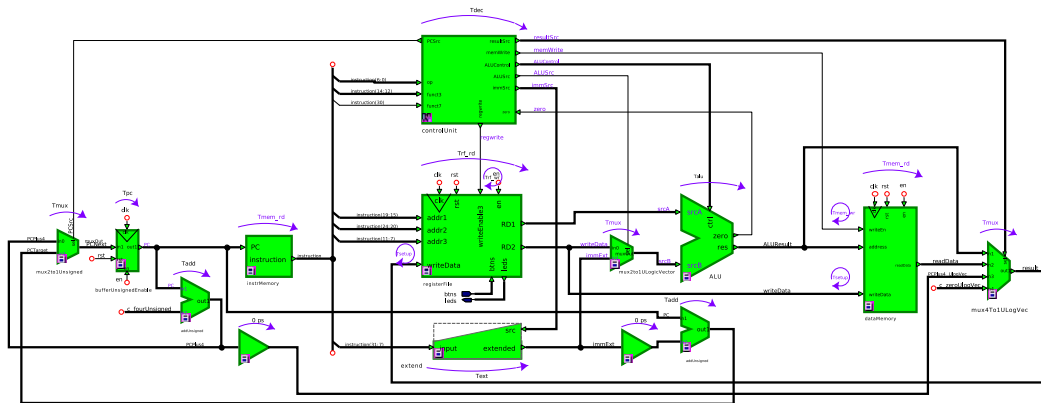


Abbildung 1: Top level

2.1 Analyse

Bestimmen Sie die verschiedenen Funktionen der Schaltung:

1. Welche Rolle spielt jeder Block?
2. Identifizieren Sie die fünf Schritte der RISC-V -Pipeline und welche Blöcke damit verbunden sind.
3. Wozu dienen die einzelnen Signale? Wie gross sind sie?



3 Control Unit

Eines der Hauptmerkmale, das einen Prozessortyp von einem anderen unterscheidet, ist die Art und Weise, wie die Befehle kodiert werden, dies hängt mit der internen Struktur zusammen. Zum Beispiel gibt es die Operation Addition, welche eine Zahl mit einem Register addiert, in der RISC-V -Architektur genauso wie in **x86** oder **ARM**, aber der erzeugte Opcode ist nicht der gleiche.

Für die Operation **addi x0 x0 1** in RISC-V (**add r0, r0, 1** in ARM) ist das Ergebnis :

- ARM : 0x 01 00 80 E2
- RISC-V : 0x 00 10 00 13

Auf der Abbildung Abbildung 2 ist der Kontrollblock abgebildet der alle benötigten Eingangssignale erhält um die verschiedenen Kontroltsignale zu generieren. Es ist derjenige, der dafür zuständig ist, die Schaltung gemäss der gegebenen Anweisung zu konfigurieren. Der **controlUnit** kann zur Vereinfachung der Schaltung noch in 2 Subblöcke **mainDecoder** sowie **ALUDecoder**:

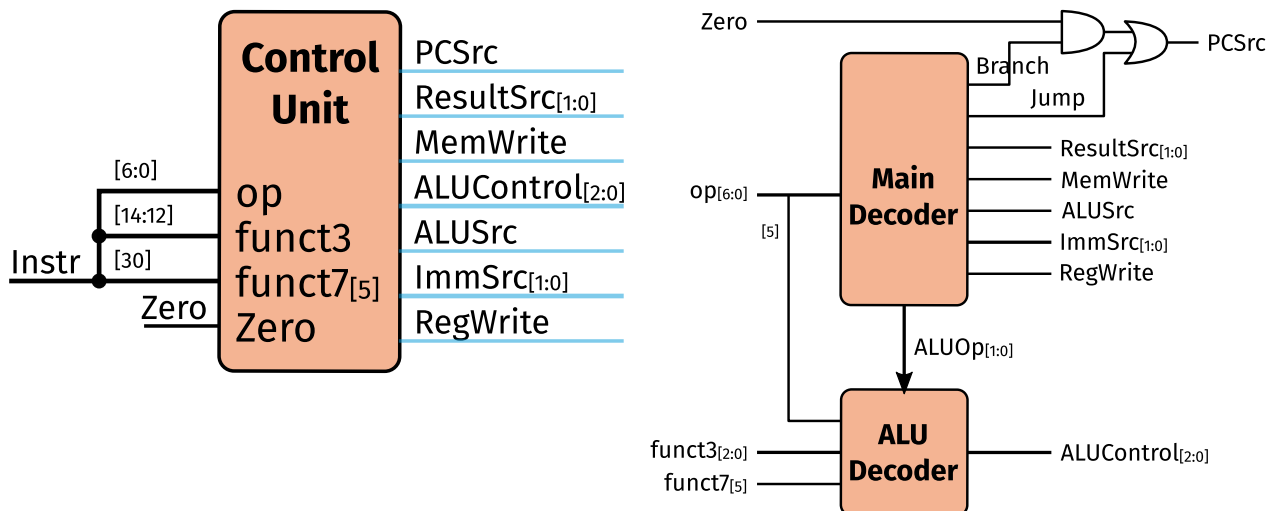


Abbildung 2: Control Unit

3.1 Umsetzung

Ergänzen Sie den Block **controlUnit**, um die folgenden Anweisungen zu unterstützen:

- R-type : add, sub, slt, or, and
- I-type : addi, slti, ori, andi, lw
- S-type : sw
- B-type : beq
- J-type : jal

Schreiben Sie hierzu für beide Subblöcke **mainDecoder** sowie **ALUDecoder** eine Wahrheitstabelle indem Sie sich auf die Tabellen Abbildung 3 und Abbildung 4 beziehen. In der Tabelle Abbildung 5 finden Sie weitere Hinweise auf das **ALUOp**-Signal, das zwischen den beiden Blöcken läuft.

Die Blöcke sind direkt in VHDL realisiert. Referenzcodes sind in den jeweiligen Blöcken verfügbar.



Um solche Blöcke in HDL Designer zu erstellen, wählen Sie **VHDL File** → **Architecture** und stellen Sie sicher, dass die Sprache auf **VHDL 2008** eingestellt ist. Auf der nächsten Seite steht **Architecture** für den Namen der Ansicht (ein Block kann verschiedene Inhalte haben) und **Entity** für den Namen des Blocks (z. B. mainDecoder).

3.1.1 Extend

Der Block extend stützt sich auf den von **immSrc** vorgegebenen Anweisungstyp, um das Vorzeichen des unmittelbaren Wertes zu extrahieren und zu erweitern:

immSrc	Type
00	I
01	S
10	B
11	J

Abbildung 3: Bestelltabelle für den Extend-Block

3.1.2 ALU

Die ALU realisiert die arithmetischen und logischen Funktionen nach dem Signal **ALUSrc** gemäß der folgenden Tabelle:

ALUControl	Operation
000	add
001	subtract
010	AND
011	OR
101	Set Lesser Than
Others	-

Abbildung 4: Bestelltabelle für den ALU-Block

3.1.3 ALUOp

Um Doppelungen zu vermeiden, wird $op_{[6:0]}$ nicht in den Block **aluDecoder** übernommen. Stattdessen erzeugt **mainDecoder** das kleinere Signal **ALUOp**, das mehrere Arten von Anweisungen zusammenfasst, die nach einem willkürlichen Code auf die gleiche Weise arbeiten. Beispielsweise führen **addi x2, x3, 30** und **add x2, x3, x4** beide eine Additionsoperation aus. Die Anweisungen I und R sind daher unter demselben Code zusammengefasst.

Die folgende Tabelle zeigt diese Idee der Vereinheitlichung und listet die speziellen Kästchen für die Dekodierung auf:



ALUOP	Funct3	Op ₅ / funct7 ₅	instr	ALUControl _[2:0]
00	---	--	LW, SW	000 (add)
01	---	--	BEQ	001 (sub)
10	000	00, 01, 10	ADD, ADDI	000 (add)
10	000	11	SUB	001 (sub)
10	010	--	SLT, SLTI	101 (slt)
10	110	--	OR, ORI	010 (or)
10	111	--	AND, ANDI	011 (and)

Abbildung 5: Dekodierungstabelle des Blocks ALUDecoder

4 Simulation

Unter dem Ordner **Simulation** finden Sie die Codes, die zur Simulation und zum Einsatz des Systems verwendet werden, in drei Formaten:

- **.s**: Assemblercode, unverschlüsselt
- **.bin**: kompilierter Code; kann mit Tools wie [Ghidra](#) oder Online-Disassemblern analysiert werden.
- **.txt**: kompilierte Version, die mit dem FPGA -Speicher für den Einsatz kompatibel ist.

Der Code wird mithilfe des Programms **HEIRV32-ASM** kompiliert, das zusammen mit der Dokumentation im gleichnamigen Ordner verfügbar ist.

Der Tester **heirv32_sc_tb** verwendet bei der Simulation den Code **codesim.bin**:

```
# Base code :
# riscvtest.s
# Sarah.Harris@unlv.edu
# David_Harris@hmc.edu
# 27 Oct 2020

main:
    addi x2, x0, 5      # x2 = 5
    addi x3, x0, 12     # x3 = 12
    addi x7, x3, -9     # x7 = 12 - 9 = 3
    or    x4, x7, x2     # x4 = (3 or 5) = 7
    and   x5, x3, x4     # x5 = (12 AND 7) = 4
    add   x5, x5, x4     # x5 = 4 + 7 = 11
    beq   x5, x7, end    # shouldn't be taken
    slt   x4, x3, x4     # x4 = (12 < 7) = 0
    beq   x4, x0, around # should be taken
    addi  x5, x0, 0      # shouldn't execute
around:
    slt   x4, x7, x2     # x4 = (3 < 5) = 1
    add   x7, x4, x5     # x7 = (1 + 11) = 12
    sub   x7, x7, x2     # x7 = (12 - 5) = 7
    sw    x7, 84(x3)     # [96] = 7
    lw    x2, 96(x0)     # x2 = [96] = 7
```



```
add x9, x2, x5      # x9 = (7 + 11) = 18
jal x3, end         # jump to end, x3 = 0x44
addi x2, x0, 1      # shouldn't execute
end:
add x2, x2, x9       # x2 = (7 + 18) = 25
sw x2, 0x20(x3)      # [100] = 25
done:
beq x2, x2, main     # infinite loop
```

4.1 Analyse

Analysieren Sie zunächst den Code **codesim.s**, um seinen Inhalt zu verstehen. Ist er ein gutes Kandidat, um Ihren System zu testen?

Bestimmen Sie die Zeit, die benötigt wird, um die Schleife des Programms einmal durchlaufen zu lassen, wenn man bedenkt, dass das System mit **50 MHz** getaktet ist für EBS3 boards, und **66 MHz** für die EBS2 boards.



Der Simulator ist standardmäßig auf EBS3 eingestellt. Um dies zu ändern, öffnen Sie den Block **heirv32_sc_tb** und klicken Sie mit der rechten Maustaste auf den Block **heirv32_sc_tester** → **Change Default View** → **testEBS2**.

Starten Sie die Simulation und verifizieren Sie, dass das System funktioniert.

Beantworten Sie folgende weiterführende Fragen:

1. Was sind die Schwächen dieser Single-Cycle-Architektur?
2. Bestimmen Sie die längste Anweisung und verfolgen Sie ihren Weg zurück.
3. Wie hoch ist die maximal denkbare Clockrate?



Die Zeiten für die einzelnen Blöcke des Systems sind im Top-Level angegeben.



5 | Einsatz (optional)

5.1 EBS3

Beim Einsatz auf dem FPGA wählt das System den Code **code_sc.*** als Quelle aus.

Analysieren und verstehen Sie den betreffenden Code. Was sollte passieren?

Die Bibliothek **Board** enthält den für den Einsatz benötigten top level. Öffnen Sie ihn und analysieren Sie die Schaltung: Wie funktionieren **en_step_n** und **en_full_n**?

Die Datei **Board/concat/car-labs.lpf** enthält die Definition und die Position der Pins des Systems. Der Anschluss der Elektronik erfolgt gemäss dem folgenden Bild:

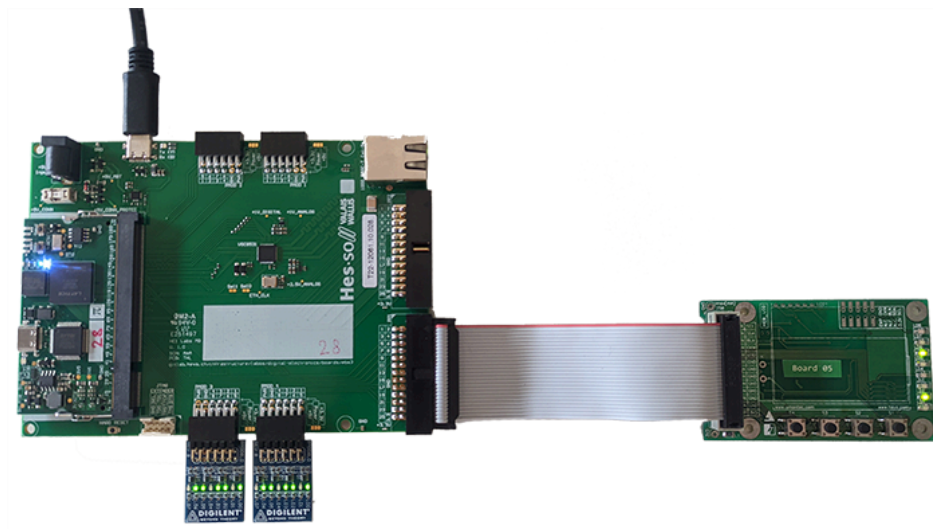


Abbildung 6: Anschliessen der Hardware

Programmieren das System auf dem FPGA und testen Sie es.



Beim Systemstart ist der Eintrag **en** von HEIRV32SC auf 0 gesetzt!



Im Dokument **BoardLFE5u-25F.pdf** finden Sie Informationen über die Synthese und den JTAG-Flash des Boards.



A | Single-cycle architecture

