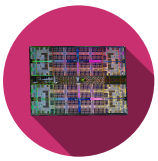


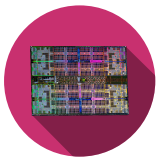
Single-Cycle RISC-V

Labor Computerarchitektur

Inhalt

1 Einführung

Dieses Labor nähert sich der Reduced Instruction Set Computer (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 Field-Programmable Gate Array (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 in Abbildung 1 dargestellt:

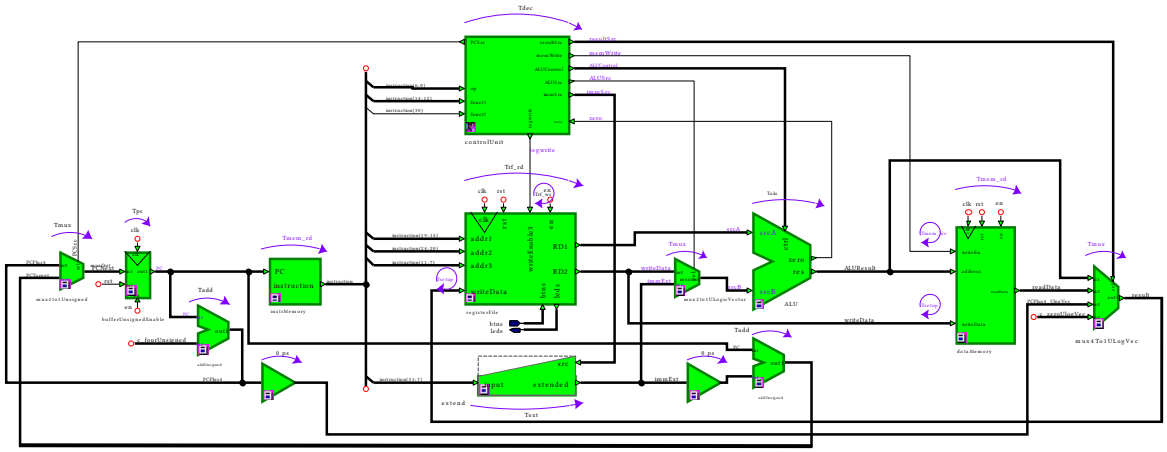


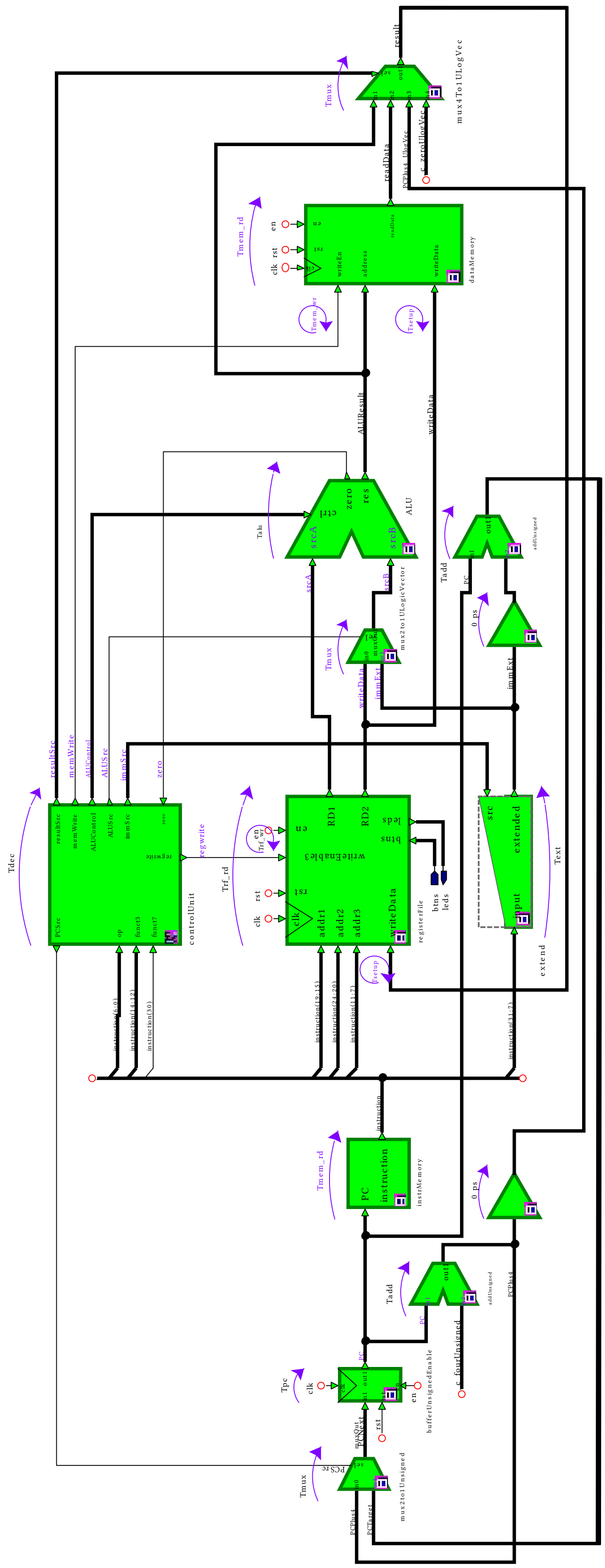
Abbildung 1 - Top level

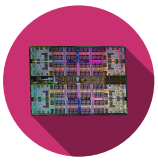
2.1 Analyse

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?

> Eine vergrösserte Version von Abbildung 1 finden Sie auf der nächsten Seite.


mux4To1ULogVec
result
readData
PCPlus4_ULogVec
c_zeroULogVec
dataMemory
Tmem_rd
Tmem_wr
Tsetup
clk
rst
en
ALUResult
writeData
ALU
srcA
srcB
ctrl
zero
res
Talu
mux2to1ULogicVector
Tmux
immExt
addUnsigned
Tadd
0 ps
resultSrc
memWrite
ALUControl
ALUSrc
immSrc
regwrite
controlUnit
Tdec
RD1
RD2
writeEnable3
addr1
addr2
addr3
writeData
btns
leds
registerFile
Trf_rd
extend
extended
Text
instruction
instrMemory
PC
PCPlus4
PCTarget
PCNext
bufferUnsignedEnable
c_fourUnsigned
Tpc
mux2to1Unsigned
PCSrc



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 Tabelle 1 ist der Kontrollblock abgebildet der alle benötigten Eingangssignale erhält um die verschiedenen Kontrolsignale 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**:

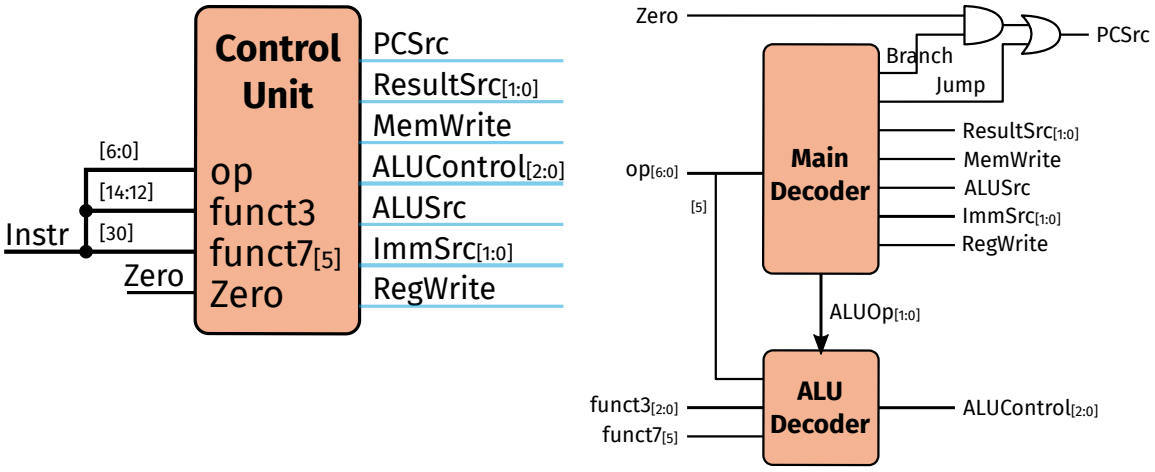


Tabelle 1 - Control Unit

3.1 Umsetzung

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

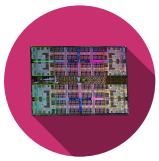
- R-type : add, sub, slt, or, and
- I-type : addi, slti, ori, andi, lw
- S-type : sw
- B-type : beq
- J-type : jal

Schreiben Sie hierzu für beide Subblöcke **mainDecoder** sowie **ALUDecoder** eine Wahrheitstabelle indem Sie sich auf die Tabellen Tabelle 2 und Tabelle 3 beziehen. In der Tabelle Tabelle 4 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.

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

Tabelle 2 - 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	**substract**
010	**AND**
011	**OR**
101	**Set Lesser Than**
Others	-

Tabelle 3 - Bestelltabelle für den ALU-Block

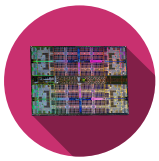
3.1.3 ALUOp

Um Duplizierungen 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	011 (or)
10	111	--	AND, ANDI	010 (and)

Tabelle 4 - 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:

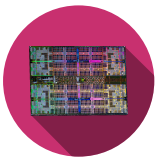
- **.s**: Assemblercode, unverschlüsselt
- **.bin**: kompilierter Code; kann mit Tools wie Ghidra oder Online-Disassemblern analysiert werden.
- **.txt**: kompilierte Version, die mit dem FPGA -Speicher für den Einsatz kompatibel ist.

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 **code_sim.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 **code_sim.s**, um seinen Inhalt zu verstehen. Ist dieser Code ein guter Kandidat, um Ihren System zu verifizieren?

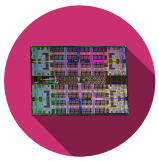
Bestimmen Sie die Zeit, die benötigt wird, um die Schleife des Programms einmal durchlaufen zu lassen, wenn man bedenkt, dass das System mit **50 MHz** getaktet ist für EBS3 boards, und **66 MHz** für die EBS2 boards.

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.



Glossar

FPGA **– Field-Programmable Gate Array** 1, 6

RISC-V **– Reduced Instruction Set Computer** 1, 2, 4