Praktikum zur Vorlesung

Prozessorarchitektur

SS 2018

Übungsblatt 7

Implementierung von Programmsteuerbefehlen in einer Befehlspipeline

Abgabefrist: Mittwoch 29.05.2018, 10:00 Uhr

1.1. Einführung

Programmsteuerbefehle sorgen dafür, dass die Befehlsabarbeitung an einer anderen Stelle der Befehlsfolge fortgesetzt werden kann. Es gibt zwei Steuerbefehlsarten: 1) Bedingte Sprünge und 2) unbedingte Sprünge. Bedingte Sprünge verzweigen nur, wenn eine bestimmte Bedingung erfüllt ist. Unbedingte Sprunganweisungen verzweigen immer. Die Zieladresse kann als absoluter Wert, der in den Programmzähler übertragen wird, abgelegt sein.

1.2. Projektbeschreibung

In dieser Übung wird die Befehlsmenge unseres Prozessors um bedingte und unbedingte Sprungbefehle erweitert. Ihre Aufgabe ist es, die Befehlspipelinearchitektur aus der vergangenen Übung so zu modifizieren, dass die neu hinzugefügten Operationen ausgeführt werden können. Die bedingten Sprungbefehle sind als I-Typ-Instruktionen wie folgt kodiert.

type (2 bit)	opcode (3 bit)	address (8 bit)	sr1 (3 bit)

Führen Sie zwei bedingte Sprungoperationen ein: **beqz** (branch if equal to zero) und **bneqz** (branch if not equal to zero). Der Prozessor prüft die Verzweigungsbedingung in Abhängigkeit vom Wert des im Befehl angegebenen Register **sr1** und setzt die Befehlsabarbeitung an dem Befehl fort, dessen Adresse in der Instruktion im **Adress**-Feld gegeben ist. Zur Vereinfachung ist in unserem Beispiel im Adress-Feld der Verzweigungsoperation eine absolute Adresse gegeben und kein Adressen-Offset zum aktuellen PC. Hier zwei Beispielbefehle:

beqz x”55”, r3 # if $r3 = 0 then PC = x”55” else PC = PC+1
bneqz x”55”, r3 # if $r3 /= 0 then PC = x”55” else PC = PC+1

Unbedingte Sprungbefehle werden im J-Typ Format wie folgt kodiert:

type (2 bit)	opcode (3 bit)	address (8 bit)	000

In dieser Übung definieren wir nur einen Sprungbefehl **jmp addr**:

jmp x”55” # jump to address x”55”

Unsere Instruktionstabelle sieht jetzt so aus:

R-Instruction [15:0]	**type [15:14]**	**opcode [13:11]**	**opcode [10:9]**	**sr1 [8:6]**	**sr2 [5:3]**	**dst [2:0]**	**Operation**
add r1, r2, r3	01	000	00	r1	r2	r3	r3 = r1 + r2
sub r1, r2, r3	01	001	00	r1	r2	r3	r3 = r1 - r2
inc r1	01	010	00	r1	x	r1	r1 = r1 + 1
dec r1	01	011	00	r1	x	r1	r1 = r1 – 1
and r1,r2,r3	01	100	00	r1	r2	r3	r3 = r1 and r2
or r1,r2,r3	01	101	00	r1	r2	r3	r3 = r1 or r2
xor r1,r2,r3	01	110	00	r1	r2	r3	r3 = r1 xor r2
not r1,r3	01	111	00	r1	x	r3	r3 = not r1
I-Instruction [15:0]	**type [15:14]**	**opcode [13:11]**	**constant [10:3]**			**sr1/dst [2:0]**	**Operation**
addi imm,r3	10	000	imm[7:6]	imm[5:3]	imm[2:0]	r3/r3	r3 = r3 + imm
subi imm,r3	10	001	imm[7:6]	imm[5:3]	imm[2:0]	r3/r3	r3 = r3 – imm
beqz addr, r3	10	010	addr[7:6]	addr[5:3]	addr[2:0]	r3/-	r3=0 -> pc=addr
bneqz addr, r3	10	011	addr[7:6]	addr[5:3]	addr[2:0]	r3/-	r3≠0 -> pc=addr
andi imm,r3	10	100	imm[7:6]	imm[5:3]	imm[2:0]	r3/r3	r3 = r3 and imm
ori imm,r3	10	101	imm[7:6]	imm[5:3]	imm[2:0]	r3/r3	r3 = r3 or imm
xori imm,r3	10	110	imm[7:6]	imm[5:3]	imm[2:0]	r3/r3	r3 = r3 xor imm
J-Instruction [15:0]	**type [15:14]**	**opcode [13:11]**	**address [10:3]**			**not used**	**Operation**
jmp addr	11	000	addr[10:3]			000	pc = addr

1.3. Schritt 1: Erstellen der VHDL-Quellen der Datenpfadblöcke

Sie werden ein neues Projekt in Vivado mit dem Namen lab7 erstellen. Alle VHDL-Quellen aus Übung 6 werden in das neue Projekt importiert. Ändern Sie die bestehenden VHDL-Quellen, so

dass sie die erforderliche Funktionalität erfüllen. Alle Logikblöcke die von der Verzweigungslogik gebraucht werden, sind in der Ausführungsphase implementiert.

1. Erstellen Sie einen kombinatorischen funktionalen Block in VHDL zum Vergleichen (*cmp.vhd*). Der Vergleicher nimmt ein 8-Bit Wort als Eingabe, prüft ob das Eingabewort gleich Null ist und liefert eine 1-Bit Ausgabe zurück. Die Ausgabe (das sogenannte *zero*-Flag) ist 1, gdw. die Eingabe gleich 0 ist.

2. Erweitern Sie die Kontrolllogik um die Funktionalität die von der Verzweigungskontrolle benötigt wird. Die Kontrolllogik soll ein 2-Bit Ausgabesignal (br_cond[1:0]) an die Ausführungseinheit weiter geben. Das Kontrollsignal weist auf eine anstehende Sprungoperation hin. Die Codierung des Verzweigungs-Kontrollsignals ist wie folgt gegeben:

br_cond[1:0]	**Anstehende Instruktion**
00	no branch/jump
01	beqz
10	bneqz
11	jmp

3. Generieren Sie einen Logik Block in VHDL zur Steuerung der Verzweigung (*br_ctrl.vhd*). Dieser erzeugt ein Lade-Signal für den Programmcounter PC. Es wird ‚1' gesetzt, wenn eine Verzweigungs-/Sprunginstruktion ansteht und die Verzweigungsbedingung erfüllt ist.

4. Modifizieren Sie den Programmcounter (PC) (*pc.vhdl*), so dass er mit der Adresse des Verzweigungsziels (8-Bit Eingabe in den PC) aktualisiert wird, wenn das Load-Signal (1-Bit Eingabe in den PC) auf ‚1' gesetzt ist.

5. Importieren Sie die aktuelle ***imem.vhd*** von der Homepage.

6. Aktualisieren Sie zum Schluss das Datenpfadmodul (datapath.vhd) mit den neuen Logikblöcken. Die Verzweigungslogik muss in die Ausführungsphase des Datenpfades eingefügt werden. Beachten Sie, dass die notwendigen Steuersignale in der Steuereinheit richtig generiert und durch die Pipeline geleitet werden.

   Das Top-Modul sollte folgende I/O-Ports haben:

   - ***clk*** – Takteingang.
   - `rst_n` – asynchrones Reset (low aktiv).
   - `pc_data[7:0]` – Ausgabedaten aus dem Programm-Counter.
   - `ir_data[15:0]` – Ausgabedaten aus dem Instruktionsregister (IF/ID Pipeline Stufe)

- `rf_data[20:0]` – Ausgabedaten aus der ID/EX Pipeline Stufe.
  rf_data = 'Dst Sel' & 'Reg A' & 'Reg B' & 'EX' & 'WB';
- `alu_data[11:0]` – Ausgabedaten aus der EX/WB Pipeline Stufe.
  alu_data = 'Dst Sel' & 'Alu Reg' & 'WB';
- `ctrl_data[9:0]` – Ausgabedaten aus der Steuereinheit.
  ctrl_data = pc_wr & ir_wr & rf_ra_sel & alu_reg_wr & alu_b_sel & rf_reg_wr & rf_dst_wr & alu_op;
- `br_data[10:0]` – Ausgabe der Ausführungseinheit für die Ausführung der Verzweigungsoperation.
  br_data = branch_condition[1:0] & branch_flag_to_pc & branch_address_to_pc[7:0]

7. Skizzieren Sie **schriftlich** die Ausführungsstufe ihrer Pipeline, indem Sie die Signalverbindungen zwischen den Blöcken herstellen.

1.4. Schritt 2: Simulieren des Top-Designs

Importieren Sie die Testbench-Datei ***datapath_tb.vhd*** und starten Sie die Simulation. Es wird ein ähnliches Programm wie in den früheren Übungen ausgeführt. Das Programm ist um einige Verzweigungs-/Sprungbefehle erweitert worden. Der Programmablauf wurde mit nop-Instruktionen erweitert um Datenabhängigkeiten zu vermeiden. Das Testbench-Programm ist im IMEM-Block gespeichert. Das vollständige Programm ist in Tabelle 1 beschrieben:

1.5. Schritt 3: Synthetisieren des Top-Designs

- Führen Sie die Synthese des Top-Moduls durch.
- Generieren Sie den 'Timing Summary Report' und überprüfen Sie die Ergebnisse für 'Setup Delay' unter 'Unconstrained Paths'. Was ist das 'Worst Setup Delay'? Was ist der Unterschied zum Single-Cycle und Multi-Cycle Datapath?

Tabelle 1. Testbench-Programm.

Adresse im Speicher	Instruktion	Aktion
00	addi 0, r0	r0 = 0
01	addi 1, r1	r1 = 1
02	addi 2, r2	r2 = 2
03	addi 3, r3	r3 = 3
04	addi 4, r4	r4 = 4
05	addi 5, r5	r5 = 5
06	addi 6, r6	r6 = 6
07	addi 7, r7	r7 = 7
08	add r0, r1, r7	r7 = 1
09	sub r5, r4, r6	r6 = 1
0A	inc r0	r0 = 1
0B	dec r2	r2 = 1
0C	and r1, r5, r5	r5 = 1
0D	xori 5, r4	r4 = 1
0E	andi 1, r3	r3 = 1
0F	or r1, r1, r1	r1 = 1
10	addi x”FE”, r0	r0 = x”FF”
11	nop	stall
12	nop	stall
13	subi 1, r0	r0 = x”FE”
14	nop	no action
15	nop	no action
16	not r0, r0	r0 = 1
17	jmp x”30”	PC = x”30”
18	nop	stall
19	nop	stall
…	…	…
20	dec r7	r7 = 1
…	…	…
30	sub r2, r3, r4	r4 = 0
31	nop	stall
32	nop	stall
33	beqz x”80”, r4	if r4 = 0, PC=x”80”, else PC+1
34	nop	stall
35	nop	stall
36	add r5, r6, r7	r7 = 2
37	nop	stall
38	nop	stall
39	bneqz x”20”, r7	if r7 /= 0, PC=x”20”, else PC+1
3A	nop	stall
3B	nop	stall
…	…	…
80	inc r4	r4 = 1
81	jmp x”36”	PC = x”36”
82	nop	stall
83	nop	stall