Rechnerarchitektur – Praktische Übungen

Übung 5: Pipelining

Aufgabe 5.1: Pipeline-Prinzip (2 Punkte)

Wenden Sie das Pipeline-Prinzip auf folgendes Beispiel an: Das Pflanzen von Stiefmütterchen im Garten bestehe aus folgenden einzelnen Tätigkeiten:

1. Auspacken des Setzlings (5 Minuten)
2. Auflockern der Erde und Graben eines Lochs (9 Minuten)
3. Einbringen von Düngestäbchen (3 Minuten)
4. Einsetzen des Setzlings und vorsichtiges Verschließen des Lochs (10 Minuten)
5. Gießen (5 Minuten)

Wie sieht der Pflanzvorgang mit und ohne Pipelining aus? Geben Sie den Zeitaufwand mit und ohne Pipelining an. Berechnen Sie die maximale Taktrate sowie die Beschleunigung durch das Pipelining.

Aufgabe 5.2: Achtstufige Pipeline (2 Punkte)

In einer achtstufigen Pipeline braucht die längste Stufe (ohne Pipeline-Register) 0,9 ns. Die Verzögerung durch die Pipeline-Register beträgt 0,1 ns. Was ist die Taktzeit (clock cycle time) für die achtstufige Pipeline?

Nun wird aus der achtstufigen Pipeline eine sechzehnstufige Pipeline indem jede Stufe halbiert wird. Bestimmen Sie die resultierende Taktzeit.

Aufgabe 5.3: Reservation Table (2 Punkte)

Führen Sie folgende aus der Vorlesung bekannte Reservation Table fort:

	4	5	6	7	8	9	10	11	12	13	14	16	16	17
`FADD yk,yk,q`	D				X	X	X	X	M	W				
`SET xk,temp1`	F				D				X	M	W			
`FMUL temp1,xk,xk`														
`FMUL temp2,yk,yk`														

Aufgabe 5.4: Umordnen der Befehlsreihenfolge (2 Punkte)

Darf bei folgenden Programmfragmenten die Reihenfolge der Befehle verändert werden? Wenn nein, warum nicht?

a)
```
FMUL temp1,xk,xk
FMUL temp2,yk,yk
```

b)
```
ADD temp2,temp2,bildx
STBU k,temp1,temp2
```

c)
```
FMUL yk,xk,yk
SETH xk,#4000        2,0 (Gleitkommawert!)
FMUL yk,yk,xk
```

d)
```
FMUL yk,xk,yk
SETH temp2,#4000     2,0 (Gleitkommawert!)
FMUL yk,yk,temp2
```

Aufgabe 5.5: Result-Forwarding und Prefetching (2 Punkte)

Welche der folgenden Aussagen (jeweils ein oder mehrere) sind richtig? Kreuzen Sie die jeweils richtigen Aussagen an.

Die Technik des Result-Forwarding…

- ☐ …hilft, die Ausführung von Sprungbefehlen auf einer Pipeline zu beschleunigen.
- ☐ …hilft, die Ausführung von arithmetischen Befehlen auf einer Pipeline zu beschleunigen.
- ☐ …ist sehr aufwändig und wird daher nur in Spezialprozessoren eingesetzt.

Die Technik des Prefetching…

- ☐ …hilft, die Ausführung von Speicherzugriffen auf einer Pipeline zu beschleunigen.
- ☐ …lädt Befehle aus dem Speicher, sofern ein Befehl durch die M-Phase leer läuft (kein Speicherzugriff).
- ☐ …ist sehr aufwändig und wird daher nur in Spezialprozessoren eingesetzt.

Aufgabe 5.6 (praktisch): OpenRISC Pipeline, Sprungvermeidung (4 Punkte)

In dieser praktischen Aufgabe soll das Sprungverhalten der OpenRISC CPU untersucht werden. Hierzu nutzen wir die von Ihnen im vorherigen Aufgabenblatt begonnene Implementierung innerhalb der Datei `hmcmd.c` und erweitern diese um Tests für das Sprungverhalten.

Hintergrundinformation: Pipeline der OR 1200 CPU

Die OpenRISC 1200 CPU implementiert die fünf Stufen einer klassischen RISC-Pipeline, wie wir sie innerhalb der Vorlesung kennengelernt haben, vergl. Abbildung 1. Um die Latenz zu verringern, werden jedoch regulär nur die Stufen Fetch, Decode, Execute und Writeback durchlaufen (d.h. es gibt regulär nur vier Pipelinestufen). Die „Memory (Access)“ Stufe wird nur bei Befehlen durchlaufen, die mit Hilfe der Load/Store Unit (LSU) auf den Speicher zugreifen. Bei diesen Befehlen wird die restliche Pipeline kurz angehalten (Pipeline Stall) während die Memory Stufe durchlaufen wird. (Innerhalb des Verilog-Quelltextes des Prozessors sind die Pipelinestufen in den Quelltextdateien `or1200_cpu.v` und `or1200_ctl.v` implementiert.)

Zur Vereinfachung können Sie bei der Erstellung der Pipelinediagramme davon ausgehen, dass alle Befehle die fünf Stufen durchlaufen – wie in der Vorlesung besprochen.

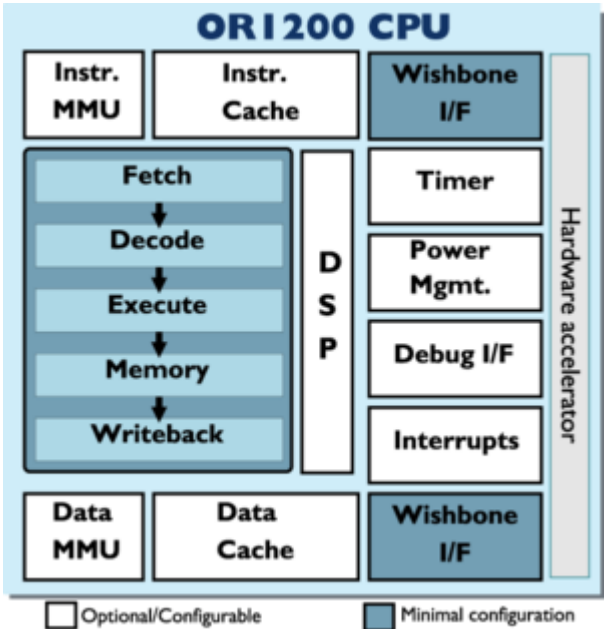

Abbildung 1: Pipeline der OpenRISC CPU (Quelle: http://opencores.org/or1k/OR1200_OpenRISC_Processor)

Schritt 1: Messung der CPI-Werte für bedingte Sprünge

Nutzen Sie das Ihnen aus dem vorhergehenden Aufgabenblatt bekannte Verfahren zur Bestimmung von CPI-Werten, um die Anzahl an benötigten CPU Takten für folgende Befehle zu bestimmen:

a) Bedingter Sprung, der durchgeführt wird (d.h. Bedingung ist wahr)

b) Bedingter Sprung, der NICHT durchgeführt wird (d.h. Bedingung ist falsch)

Implementieren Sie dazu einen neuen Befehl `hmcmd branch` welcher die beiden Werte misst und ausgibt. Beachten Sie dabei, dass der OpenRISC 1200 einen Delay-Slot nach bedingten Sprungbefehlen nutzt und lesen Sie die Hinweise zu (lokalen) **Sprungmarken/Labels** im Einschub „Inline-Assembler für Fortgeschrittene“.

Einschub: Inline-Assembler für Fortgeschrittene

Aus dem letzten Aufgabenblatt ist Ihnen die Nutzung von Inline-Assembleranweisungen grundsätzlich bekannt – im Folgenden werden noch einige wichtige Erweiterungen zur Einbettung in den umliegenden C-Programmcode betrachtet.

- Wenn Sie einen ganzen **Block von Inline-Assembler Anweisungen** einfügen wollen, so können Sie die einzelnen Assemblerbefehle nacheinander einfügen mit einem Newline trennen. Beispielsweise fügen Sie folgendermaßen drei `l.nop` Befehle hintereinander ein:

```
asm volatile (
"l.nop    # erster Befehl\n"
"l.nop    # zweiter Befehl\n"
"l.nop    # dritter Befehl\n"
);
```

- Wenn Sie **Daten zwischen dem C-Programm und dem Assemblerblock austauschen** wollen, so können Sie Ein- und Ausgabe Operanden angeben. Auf diese können Sie sich dann innerhalb des Assemblerteils mit Hilfe der Platzhalter %0, %1, usw. beziehen – die Operanden werden dabei in der Reihenfolge entsprechend durchnummeriert.

```
asm ( "Assemblerbefehltemplate mit Platzhaltern"
          : Ausgabeoperanden                /* optional */
          : Eingabeoperanden                /* optional */
          : Liste modifizierter Register    /* optional */
          );
```

  Beispiel:

```
int ergebnis;
int eingabe = 42;
asm (
  "l.addi %0,%1,3      # addiere 3 zur Eingabe\n“
  : "=r" (ergebnis) : "r" (eingabe)
);
```

  Hierbei teilt das Constraint "r" dem Compiler mit, dass er ein Register zur Übergabe verwenden soll und das "=" zeigt an, dass das die entsprechende Übergabevariable nur beschrieben wird.

- **Lokale Sprungmarken** (lokale Labels): Wenn Sie innerhalb eines Assemblerblocks eine lokale Sprungmarke definieren wollen, so können Sie dieses mit Hilfe einer ganzzahligen Zahl und einem nachgestellten Doppelpunkt bewirken. Wenn Sie das entsprechende Label anspringen wollen, so müssen Sie jeweils angeben, ob der Sprung nach vorne (forward, „f“ an das Label anhängen) oder nach hinten (backward, „b“ an das Label anhängen) ausgeführt wird.

  Beispiel:

```
asm (
  […]
"l.bnf 3f        # wenn Flag nicht gesetzt, springe nach vorne zur lokalen Sprungmarke 3\n“
  […]
 "3:             # lokale Sprungmarke Nummer 3\n“
  […]
 "l.bnf 3b       # wenn Flag nicht gesetzt, springe zurück zur lokalen Sprungmarke 3\n“
  […]
);
```

Ausgewählte Befehle der OpenRISC 1200 CPU

Einige der für diese Aufgabe unter Umständen relevanten Befehle aus dem Basic Instruction Set der OpenRISC CPU finden Sie in untenstehender Tabelle.

Befehl	Bedeutung
`l.addi rD,rA, I`	„add immidiate“ – Konstante I wird zu rA addiert, Ergebnis in rD gespeichert.
`l.lwz rD, I(rA)`	Lädt ein Wort von der Adresse rA + Konstante `I`. `I` kann 0 sein.
`l.sfeqi rA,I`	„Set flag if equal to immidate“: vergleicht rA mit Konstante I und setzt Compare Flag, falls rA gleich I ist.
`l.sfgtsi rA,I`	„Set flag if greater immidiate, signed“: vergleicht rA inkl. Vorzeichen mit Konstante I und setzt Compare Flag, falls rA größer als I ist.
`l.bf N`	„Branch if flag“: Bedingter Sprung - falls flag gesetzt ist, Sprung zu Adresse N.
`l.bnf N`	„Branch if flag“: Bedingter Sprung - falls flag NICHT gesetzt ist.
`l.cmov rD,rA,rB`	“Conditional Move”: Falls Flag gesetzt, so wird rA in rD kopiert, andernfalls wird rB in rD kopiert.

Eine vollständige Referenz finden Sie unter
http://opencores.org/websvn,filedetails?repname=openrisc&path=%2Fopenrisc%2Ftrunk%2Fdocs%2Fopenrisc-arch-1.0-rev0.pdf im OpenRISC 1000 Architecture Manual.

Schritt 2: Laufzeitmessung einer Zählschleife

Das folgende Assembler-Fragment zählt die Anzahl positiver Werte innerhalb eines Integer-Arrays:

```
#define NR_ELEMENTS 100
#define STRINGI(x) #x
#define STR(x) STRINGI(x)
void do_count()
{
        int input[NR_ELEMENTS];

        […]

        for(i=0; i<runs; i++){

                asm volatile (
                "l.addi r5,r0,1                 # set r5 to 0 - this is the loop counter\n"
                "l.addi %0,r0,0                 # set result to 0\n"
                "l.addi r6,%1,0                 # put start address of input in r6\n"

                "0:                             # start of loop\n"
                "l.lwz r7,0(r6)                 # load next value to be checked\n"
                "l.sfgtsi r7,0                  # set flag if >0\n"
                "l.bnf 1f                       # if not positive, cont. loop without incrementing\n"
                "l.addi r6,r6,4                 # increment position in memory (delay slot!)\n"
                "l.addi %0,%0,1                 # increment number of counted positives\n"

                "1:                             # continue loop...\n"
                "l.sfeqi r5, " STR(NR_ELEMENTS-1) " # check if end of loop reached\n"
                "l.bnf 0b                       # if not all iterations are done, do another loop\n"
                "l.addi r5,r5,1                 # increment loop counter (delay slot!)\n"
                : "=r" (result) : "r" (input) : "r5","r6","r7");
        }

        printf("Number of positive elements: %i\n", result);
        […]
}
```

Befüllen Sie das Array `input` mit einer Mischung aus positiven und negativen Werten. Bestimmen Sie dann die Anzahl an CPU-Takten, welche für den Durchlauf des Assembler-Fragments benötigt werden. Geben Sie das Ergebnis der Messung aus. Wie wirkt sich das Verhältnis von positiven zu negativen Werten auf die Laufzeit aus?

Anzahl benötigter CPU-Takte: _____

Geben Sie die Reservation Table für einen Schleifendurchlauf an. (Falls Sie die Tabellen nicht selbst zeichnen wollen, so finden Sie ein Excel-Template dafür auf Moodle.)

Schritt 3: Sprungvermeidung durch bedingte Befehle

Schreiben Sie das Code-Fragment durch Nutzung des bedingten Befehls `l.cmov` so um, dass der Sprung innerhalb des Schleifenkörpers nicht mehr notwendig ist. Messen Sie erneut die Anzahl an benötigten CPU-Takten. Geben Sie das Ergebnis der Messung aus.

Anzahl benötigter CPU-Takte: _____

Schritt 4: Sprungvermeidung durch (teilweises) Ausrollen der Schleife (partial loop unrolling)

Schreiben Sie das Code-Fragment aus Schritt 3 so um, dass nur noch ein Viertel der Anzahl der Schleifendurchläufe benötigt wird. (Annahme: NR_ELEMENTS ist immer ein Vielfaches von 4.) Pro Schleifendurchlauf müssen demnach vier Werte geprüft werden. Messen Sie erneut die Anzahl der benötigten CPU-Takte und geben Sie das Ergebnis der Messung aus.

Anzahl benötigter CPU-Takte: _____

Geben Sie die Reservation Table für einen Schleifendurchlauf an. (Falls Sie die Tabellen nicht selbst zeichnen wollen, so finden Sie ein Excel-Template dafür auf Moodle.)

Für die Abnahme dieser praktischen Aufgabe brauchen Sie:

- das lauffähige, von Ihnen modifizierte orpmon-Programm mit den Messungen von Schritt 1-4,
- den zugehörigen, übersetzbaren Quellcode,
- die Messwerte für die Schritte 1-4
- die Reservation Tables aus Schritt 2 und Schritt 4