RA FS 18 Lösung Repetitionsserie: Teil MIPS

Qiyang Hu, Givi Meishvili, Adrian Wälchli, Mauro Kiener

2 MIPS

2.1 Grosse Konstanten

Warum ist es nicht möglich, mit dem 32-Bit MIPS-Instruction Set eine Konstante der Länge 32 Bit mit genau einem Maschinenbefehl zu laden?

Lösung:

Jeder Befehl hat eine feste Länge von 32 Bit. Da die ersten 6 Bit davon vom `opcode` belegt werden, ist es nicht möglich eine Konstante von 32 Bit mit einem Befehl zu laden.

2.2 Jump

Erklären Sie den Unterschied zwischen dem Befehl `jal` und dem Befehl `j`.

Lösung:

Beim Befehl `j` wird zum angegebenen Label gesprungen. Das nächste Sprungziel muss im folgenden Code definiert werden.

Der Befehl `jal` wird für Funktionsaufrufe verwendet. Der Befehl springt zur Subroutine und speichert zusätzlich das Rücksprungziel in Register `$ra`.

2.3 MIPS ISA

Schreiben Sie den folgenden C-Programmauschnitt in Assembler um:

```
/* ...something... */
array[4] += 1638410; // Datentyp des Arrays: int
/* ...something... */
```

Nehmen Sie an, dass sich die Startadresse des Arrays im Register `$s1` befindet und dass es sich um eine 32-Bit Architektur handelt.

Lösung:

```
1 lw $s2, 16($s1) # Adresse mit Offset laden
2 lui $t0, 25 # Oberen Teil der Konstante laden
3 ori $t0, $t0, 10 # Unteren Teil der Konstante laden
4 add $s2, $s2, $t0 # Zahlen zusammenzaehlen ...
5 sw $s2, 16($s1) # ... und zurueckschreiben
```

2.4 MIPS Multiplikation

Schreiben Sie den folgenden C-Programmauschnitt in Assembler um, verwenden Sie dabei ausschliesslich Additionsbefehle und (bedingte und unbedingte) Verzweigungen.

Nehmen Sie an, `i` stehe in Register `$s2` und `j` in Register `$s1` und dass `i` und `j` positive Ganzzahlwerte enthalten.

```
/* ...something... */
i *= j;
/* ...a bit more of something ...*/
```

Lösung:

```
1       addi $t0, $zero, 1 # int count = 1; ( i*=j iff i=i*j iff i=i+i*(j-1) )
2       add $t1, $s2, $zero # int t = i;
3  MULT: beq $s1, $t0, CONT # while(j != count);
4       add $s2, $s2, $t1 # i = i + t;
5       addi $t0, $t0, 1 # count++
6       j MULT
7  CONT:
```

2.5 $at Register

Wozu gibt es das Register `$at`, welches für den Assembler reserviert ist?

Lösung:

Das Register `$at` vereinfacht die Umsetzung von Pesudobefehlen, wie z.B. `bgt`, `bge`, `blt` oder `ble`. Der Befehl

```
1  bgt $rs, $rt, label
```

wird zu

```
1  slt $at, $rt, $rs
2  bne $at, $zero, label
```

Ohne `$at` müsste das Resultat von `slt` in einem anderen Register gespeichert werden, welches zuerst gesichert werden müsste.

2.6 Noch mehr grosse Konstanten

Um einen festen 32bit Wert in ein Register zu laden, braucht es die beiden Befehle `lui` und `ori`. Kann man anstelle von `ori` auch `addi` verwenden? Begründen Sie Ihre Antwort!

Lösung:

Statt `ori` kann nicht `addi` verwendet werden, da die `immediate`-Werte bei `addi` als vorzeichenbehaftet aufgefasst werden, d.h. ein `immediate`-Wert, welcher eine 1 als höchstwertiges Bit hat, würde zu einer Subtraktion führen und damit nicht das gleiche Resultat ergeben, wie ein `ori`.

2.7 Singlecycle vs. Multicycle

Erläutern Sie die Unterschiede zwischen Singlecycle und Multicycle Datapath Approach.

Lösung:

Singlecycle Datapath Approach: Ein Befehl pro Takt wird ausgeführt, deshalb muss sich der Takt nach dem langsamsten Befehl richten und jeder schnellere Befehl verschwendet Zeit. Da innerhalb eines Taktes jede Einheit nur einmal gebraucht werden kann, werden getrennte Instruction- und Data-Speicher benötigt[1] und zusätzlich zur ALU noch zwei Addierer. Dafür ist der Aufbau einfach zu verstehen und auch die Kontrolleinheit lässt sich entsprechend einfach gestalten.

Multicycle Datapath Approach: Aufteilen der Befehle in Teilbefehle, in jedem Takt wird ein Teilbefehl ausgeführt. Dadurch können kürzere Befehle in weniger Taktzyklen erledigt werden. Ausserdem können Einheiten mehrfach pro Befehl verwendet werden, solange dies nicht im gleichen Taktzyklus ist. Es reicht deshalb aus, ein Speicher und eine ALU zu verbauen — man spart einen Speicher und zwei Addierer ein. Dafür werden zusätzliche Register und Multiplexer benötigt, und die Kontrolleinheit wird komplexer (endlicher Automat), was insgesamt dazu führt dass die längsten Befehle etwas länger dauern als auf einer Singlecycle Implementation, im Durchschnitt wird jedoch die Ausführungszeit kürzer. Zusätzlich ist es später möglich, basierend auf dieser Variante Pipelining zu implementieren, was den Befehlsdurchsatz (CPI: Takte pro Instruktion) deutlich steigern kann.

[1] zumindest in der einfachst-möglichen Implementation

2.8 Sprünge

Weshalb ersetzt der Assembler in einigen Fällen den Befehl

```
1 beq $s0, $s1, L1
```

durch die Befehlssequenz

```
1 bne $s0, $s1, L2
2 j L1
3 L2:
```

Lösung:

Verzweigungen sind lokal, d.h. relativ zum aktuellen `PC` und daher ist die Sprungweite auf -2^15 bis $2^15 - 1$ beschränkt, da die Sprungweite durch die Grösse des `immediate`-Wertes gegeben ist (16 Bit).

`j` hingegen erlaubt weitere Sprünge, da der `address`-Wert eine Länge von 26 Bit hat.

Ist das Sprungziel also zu weit für eine Verzweigung entfernt, so ersetzt der Assembler die Befehle entsprechend.

2.9 Schleifen

Gegeben sei folgende for-Schleife (`i` ist ein Integer und `a` ist ein Integerarray):

```
1 for (i =0; i<3; i+=1) {a[i]=0}
```

(a) Wie lautet der Assemblercode, der vom Compiler erzeugt wird? Nehmen Sie an die Adresse vom Array `a` sei im Register `$s0` abgespeichert. Der Assemblercode muss die Schleife mithilfe eines Sprungbefehls implementieren.

(b) Welcher Befehl aus dem Assemblercode der vorherigen Teilaufgabe wird vom Compiler in den Delayslot gelegt, wenn der Code auf einer Pipeline- Architektur laufen soll?

Lösung:

(a)
```
1       addi $t0, $zero, 0 // offset=0
2       addi $t1, $zero, 12 // max = 3 * 4
3 loop: add $t2, $s0, $t0 // address = a + offset = i * 4
4       stw $zero, 0($t2) // a[i] = 0
5       addi $t0, $t0, 4 // offset += 4 entspricht i+=1
6       blt $t0, $t1, loop // offset < max ?
```

(b) Der `stw`-Befehl, es handelt sich hierbei um eine sogenannte "safe instruction", d.h. die Ausführung dieser Instruktion hat keine Abhängigkeiten mit dem Verzweigungsbefehl.

2.10 Jump Register

Wie muss man die MIPS Single-Cycle Architektur aus der Vorlesung erweitern, damit der `jr` Befehl unterstützt wird?

Lösung:

Es genügt, einen weiteren Multiplexer hinzuzufügen, der auswählt, ob der berechnete `PC` verwendet wird, oder ob "Read Data 1" als neuer `PC` geschrieben werden soll. Die entsprechende Steuerung geht von der Control Unit aus.

2.11 Page mode

Beschreiben Sie wie ein DRAM im Page Mode arbeitet.

Lösung:

Beim Seitenmodus wird jeweils eine komplette Reihe ($N \times M$ Bits, wobei M =Anzahl Bits eines einzelnen Outputs) vom DRAM in ein SRAM geladen, wo diese verbleiben, bis die Reihenadresse geändert wird. Nachdem die Reihe in das SRAM geladen wurde, kann via Spaltenadresse auf die einzelnen Spalten zugegriffen werden.