**Rechnerarchitektur Serie 3** Patrick Stöckli  
 16-942-468

Hoeun Yu

***Aufgabe 1: Performance Berechnung***

***(a)***

500 MHZ clock rate => 2 nsec clock cycle

Beispielsweise dauert die ALU-Operation 4 nsec. Dies bedeutet, dass die ALU-Operation 2 clock cycles benötigt (4/2 = 2). Die restlichen Operationen werden analog berechnet.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Op | Freq | CPI (Original) | Freq \* CPI (Original) | CPI (LOAD angepasst) | Freq \* CPI (LOAD angepasst) | CPI (STORE angepasst) | Freq \* CPI (STORE angepasst) |
| ALU | 25% | 2 | 0.5 | 2 | 0.5 | 2 | 0.5 |
| LOAD | 25% | 4 | 1 | 6 | 1.5 | 4 | 1 |
| STORE | 25% | 6 | 1.5 | 6 | 1.5 | 3 | 0.75 |
| BRANCH | 25% | 3 | 0.75 | 3 | 0.75 | 3 | 0.75 |
|  |  |  | 3.75 |  | 4.25 |  | 3.0 |

Dauert die LOAD-Operation 6 clock cycle, so ist die CPU um etwa 11.8% () langsamer.

Dauert die STORE-Operation nur halb so lang, so ist die CPU um 25.0% ( schneller.

***(b)***

|  |  |  |  |
| --- | --- | --- | --- |
| Op | Freq | CPI (Original) | Freq \* CPI (Original) |
| ALU | 50% | 2 | 1 |
| LOAD | 15% | 4 | 0.6 |
| STORE | 25% | 6 | 1.5 |
| BRANCH | 10% | 3 | 0.3 |
|  |  |  | 3.4 |

Die optimale CPI der CPU beträgt also 3.4.

***Aufgabe 2: Stackverwendung bei Subroutinen***

Insbesondere wenn eine Subroutine weitere Subroutinen aufruft und generell viele Parameter übergeben werden müssen, kann es sein, dass der Registerspeicher knapp werden kann. In einem solchen Falle kann ein Stack aushelfen. Die benötigten Parameter können anhand eines Stacks vorbereitet und später wenn nötig in die Register eingelesen werden. Da Parameter in einer bestimmten Reihenfolge benötigt werden, eignet sich ein Stack sehr gut.

Eine weitere Möglichkeit der Stackverwendung ist das Adressen-Handling. Dies kommt insbesondere zum Tragen, wenn die Subroutine weitere Subroutinen aufruft. Diese Subroutinen verursachen wiederum Operationen, welche eine return-Adresse haben. Mithilfe eines Stacks kann quasi eine Kette von return-Adressen gebildet werden, die nach und nach abgearbeitet wird. Damit kann auch verhindert werden, dass eine return-Adresse einer Subroutine verloren geht/ überschrieben wird.

***Aufgabe 3: ALU & SLT***

Die slt-Operation liefert boolesche Zahl zurück, nämlich 1 falls A kleiner ist als B oder 0 falls A grösser oder gleich B ist. In allen ALUs werden zudem alle Operationen gerechnet, der OP-code bestimmt dann, welches Resultat weitergeleitet wird. Daher wird auch eine Addition berechnet, selbst wenn wir das Resultat gar nicht haben möchten. Die 32. ALU ist etwas anders aufgebaut als die restlichen ALU, nämlich hat sie eine zusätzliche Leitung nach der Addition-Operation, die einen Abzweiger zu einer Set-Leitung darstellt.

Die slt-Funktion in der ALU verwendet als Basis die Subtraktion. Dies bedeutet, dass der B-Input invertiert und anschliessend mit A addiert wird. Also A + (-B). Jedoch ist nicht der Output der Subtraktion entscheidend, sondern nur das signifikanteste bit des Ergebnisses. Dies, da das signifikanteste bit das Vorzeichenbit darstellt, welches 0 ist für eine positive Zahl (und 0) und 1 für eine negative Zahl ist. Dieses Vorzeichenbit entspricht also genau der slt-Operation, da das Resultat nur negativ ist, wenn B grösser ist als A.   
Da wir aber das Ergebnis als boolesche Zahl haben möchten, wird das Vorzeichenbit via der beschriebenen set-Leitung als less-Input der ersten ALU verwendet. Bei den anderen 31 ALUs wird der less-Input auf 0 gesetzt. Durch den OP-code wählt nun der Multiplexer den Wert des less-bit aus. Dieses ist folglich in der ersten ALU der Wert des Vorzeichens, während der Wert bei den anderen 31 ALUs 0 ausgibt.

***Aufgabe 4: Loadi***

Da einzelne MIPS-Operationen keine 32-bit-Konstanten unterstützen (da nur 32 bit Platz für gesamte Operation), muss die Loadi-Operation in 2 Schritten durchgeführt werden und sieht dementsprechend folgendermassen aus:

lui $s0, <Wert[31:16]>

ori $s0, $s0, <Wert[15:0]>

Das Register $s0 wird als Beispiel genannt, es kann ein beliebiges Register genommen werden, solange das Register bei allen Instruktionsteilen gleich bleibt.  
Die lui-Instruktion lädt den Wert der oberen 16 bit (bit 16 bis 31) in den Speicher. Der in <Wert[31:16]> angegebene Dezimalwert muss diesem Wert entsprechen, nicht der des Gesamtwertes. Analoges gilt für die ori-Instruktion: es wird der Wert der unteren 16 bit der Konstanten in den Speicher geladen. Auch hier muss der in <Wert[15:0]> angegebene Dezimalwert dem Wert der unteren 16 bit (bit 0 bis 15) entsprechen. (z.B. man möchte den Wert 0000 0000 0000 0001 0000 0000 0000 0001 im Speicher haben. Dies entspricht in Dezimalzahlen dem Wert 1048577. Für die beiden Instruktionen müssen aber beide Male der Dezimalwert 1 verwendet werden, da dies jeweils dem Wert der oberen und unteren 16 bit entspricht.)

***Aufgabe 5: ALU: OPCodes***

Die Ansteuerbits müssen folgendermassen für die folgenden Operationen gesetzt werden:

* and: 0000
* or: 0001
* add: 0010
* subtract: 0110
* slt: 0111
* nor: 1100

Eine ALU hat vier Steuereingänge, weshalb die OPCodes eine Grösse von 4 bits besitzen. Die ersten zwei bits des Codes bestimmen, ob die Eingabe A und B invertiert wird oder nicht (00 = keine Eingabe wird invertiert, 01 = Eingabe B wird invertiert). Dies, weil diese beiden bits die Inverter-Multiplexer der ALU ansprechen.   
Die hinteren 2 bits sprechen den Operationen-Multiplexer an, der bestimmt, welches Ergebnis als das Resultat aus der ALU ausgegeben wird. 00 leitet das Ergebnis der UND-Operation des ALU weiter, 01 das Ergebnis der ODER-Operation, 10 das Ergebnis der ADD-Operation und 11 das Ergebnis der LESS-Operation (ist 0 wenn A>=B, ansonsten 1).  
Durch diese Grundoperationen lassen sich die anderen Operationen zusammenbauen. So ist beispielsweise subtract nichts anderes als eine Addition (10 als Input für Operation-Multiplexer) mit einem invertierten B-Input (01 für die Inverter). Dies resultiert in der Gleichung A + (-B) = A – B, was exakt das ist, was wir wollten.