Praktikum Rechnerarchitektur Sommersemester 2024

Praktikum 3

Ausgabedatum: 06.05.2024

Dieses Mal vertiefen wir unser Verständnis der MIPS ISA weiter und programmieren Fließkommaoperationen und die Auswirkungen von Pipelining auf die ISA.

In der Vorlesung haben wir gelernt, wie man die vier Grundrechenarten mit Fließkommazahlen implementieren kann. Heutige Prozessoren verwenden diese einfachen Verfahren für Addition, Subtraktion und Multiplikation, für die Division aber wie auch erwähnt, trickreiche Algorithmen wie z.B. den SRT Algorithmus.

Neben den Grundrechenarten gibt es aber natürlich auch andere mathematische Operationen, die wir mit Fließkommazahlen durchführen wollen. Diese sind typischerweise heutzutage auch in Hardware implementiert, wobei wiederum teilweise sehr trickreiche Verfahren umgesetzt werden.
Auch Softwareentwickler verwenden oftmals spezielle Algorithmen um ihre Operationen zu beschleunigen, insbesondere wenn diese Programme auf vielen Plattformen effizient laufen sollen.

Ihre Aufgabe besteht dieses Mal darin, ein Verfahren zur Quadratwurzelberechnung in MIPS Code zu implementieren, das unter dem Namen Newton-Rawles bekannt ist.

Aufgabe 1:
Verwenden Sie das C Programm unten als Vorlage um ein MIPS Programm zu schreiben, das eine Fließkommazahl einliest und dann deren Quadratwurzel berechnet und ausgibt.

Also Sie implementieren wieder eine `main` Funktion und eine `naive_sqrt` Mips Funktion. Gemäß der MIPS ABI verwendne Sie das Fließkommaregister, $f12 für die Übergabe von Gleitkommawerten mit einfacher Genauigkeit und für das Ergebnis verwenden Sie $f0.[1]

```c
#include<stdio.h>

float naive_sqrt(float x) {
	float result = 1.0;  // Floating point register dafür verwenden
	float delta;  // auch hierfür

	/* Wir iterieren solange nach Newton-Rawles, bis sich das
	 * Ergebnis  nicht mehr viel ändert (0.00001)
	 */
```

[1] Hat eine Funktion mehr als ein Fließkommaargument oder ist dies ein double-Wert, werden die Argument auf dem Stack übergeben.

```
	do {
		float temp = (x / result + result) * 0.5;
		delta = temp - result;  // auch das delta in einem fp Register
		if (delta < 0.0) {
			delta = -delta;  // sicher stellen das wir den Betrag
nehmen
		}
		// Mit neuer Näherung weiter iterieren:
		result = temp;
	} while (delta > 0.00001);

	return result;
}

int main() {
	float x = 0.0;
	float wurzel;

	printf("Geben Sie eine nicht-negative Zahl ein!\n");
	scanf("%f", &x);
	wurzel = naive_sqrt(x);
	printf("Wurzel %f = %f quadriert = %f\n", x, wurzel, wurzel * wurzel
);

	return 0;
}
```

OPTIONALE ERWEITERUNG

Zur weiteren Übung dürfen Sie auch den folgenden super-schnellen Algorithmus implementieren, der z.B. in manchen Spielen zur schnellen Bildberechnung angewandt wird.

Hier ist er in C Version:

```
float fancy_sqrt(float x) {
	const float three_halfs = 1.5F;
	long i;
	float x_half, y;

	x_half = x * 0.5F;
	y = x;

	/* Ein Hack in C, einfach in Assembler:
	 * wir interpretieren die IEEE 32 Bit Fließkommazahl als long, dh.
	 * wir nehmen einfach die Bits und kopieren Sie in ein long.
	 * Kann man in MIPS Code so machen wie hier, also über eine Variable,
	 * die im RAM ist, oder  via mfc1 um das fp Register in das
	 * Integer Register zu bekommen.
	 */
	i =  * (long *) &y;

	// Jetzt kommt die Magie des Fast Inverse Square Root Algorithmus
	// Siehe https://en.wikipedia.org/wiki/Fast_inverse_square_root
	i = 0x5f3759df - (i >> 1);


	/* Und jetzt machen wir wieder eine Fließkommazahl draus
	 * In C ist ein Hack notwendig, in Assembler einfach ein move
```

```
        * von einem Integer (long) Register in ein Float register.
        * Sie können hier also auch via RAM gehen, oder effizienter via
        * mtc1 vom integer register ins fp Register
        */
        y = * (float *) &i;


        /* Jetzt kommen noch 2 Iterationen des Newtonschen Approximations-
        * algorithmus https://en.wikipedia.org/wiki/Newton%27s_method
        */
        y = y * (three_halfs - (x_half * y * y));  // 1st iteration
        y = y * (three_halfs - (x_half * y * y));  // 2nd iteration

        /*
        * Wir wollen hier den Wurzel-Wert nicht das Inverse.
        */
        return 1.0 / y;
}
```

Aufgabe 2:

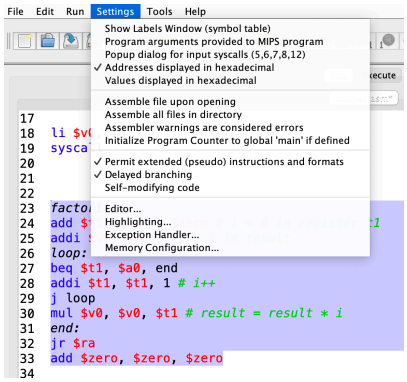
In der Vorlesung haben wir *Pipelining* als eine Art von ILP (= Instruction Level Parallelism, dt. Befehlsebenenparallelität) zur Erhöhung des Durchsatzes von Prozessoren kennengelernt. Damit das gut funktioniert, muss die Pipeline aber immer gut gefüllt sein. Sprünge stellen dabei ein Problem dar (Stichwort „control hazard“), so dass RISC Prozessoren oftmals eine sogenannten *Delayed Branch* implementieren. Da der Takt der auf den Verzweigungsbefehl folgt bei der fünfstufigen MIPS Pipeline bei einer ausgeführten Verzweigung (oder Sprung) verschwendet wäre, wird beim Delayed Branch der Folgebefehlt auf eine Verzweigung (oder auch einen unbedingten Sprung) **immer** ausgeführt, also auch wenn gesprungen wird.

Hier ist eine Implementierung der Fakultätsfunktion (n!) die dies ausnutzt, das Argument, wird natürlich via $a0 übergeben:

```
factorial:
add $t1, $zero, $zero # i = 0 in register t1
addi $v0, $zero, 1 # 1 in result
loop:
beq $t1, $a0, end
addi $t1, $t1, 1 # i++
j loop
mul $v0, $v0, $t1 # result = result * i
end:
jr $ra
add $zero, $zero, $zero
```

In ihr führen wir die Multiplikation im Delay Slot des Springes aus und die Addition der Schleifenvariablen im Delay Slot der Verzweigung.

Ergänzen Sie den Code um einen Main Funktion, die einen Integer einliest, damit die factorial Funktion aufruft und das Ergebnis dann ausdruckt. Damit der MARS Assembler die MIPS mit dem Delayed Branching Modus ausführt, müssen Sie diesen Modus unter Settings aktivieren:



Beachten Sie, dass jal, als Sprungbefehl den nachfolgenden Befehl sofort (beim Sprung ausführt!).