Institut für Informatik
Lehrstuhl für Technische Informatik

Digitale Systeme 2022

C-Programmierprojekt

1 Aufgabenstellung

Ihre Aufgabe besteht darin, einen Prozessoremulator mit der Programmiersprache C zu entwickeln. Der Prozessoremulator soll Maschinenbefehle der RISC-V Architektur nachbilden. Anschließend soll auf dem Emulator ein RISC-V Programm ausgeführt werden.

In digitalen Rechenanlagen werden Informationen ausschließlich durch zwei Kennzustände dargestellt. Diese beiden Kennzustände werden durch die Ziffern 0 und 1 symbolisiert und werden als „Binärziffer (binary digit)“ oder vereinfacht als „Bit“ bezeichnet. Als Datenwort oder Binärwort wird eine Folge von n Bit bezeichnet, die ein Prozessor in seiner arithmetisch-logischen Einheit verarbeiten kann. Bei einem Prozessor, der den RV32I Befehlssatz versteht, bestehen die Datenworte aus 32 Bit.

Computer gehorchen bedingungslos unseren Kommandos, die als „Maschinenbefehle“ bezeichnet werden. Maschinenbefehle sind genauso wie Datenworte ebenfalls eine Folge von Bits, die der Prozessor interpretiert und daraufhin eine Anweisung ausführt. Für die Ausführung eines Programms beherrschen Prozessoren eine Menge an Maschinenbefehlen, diese Menge wird Maschinenbefehlssatz „Instruction Set Architecture“ (oder kurz: ISA) genannt. Im Maschinenbefehlssatz sind Befehle enthalten, um Datenworte z.B. zu addieren, bitweise logische Verknüpfungen zwischen Datenworten durchzuführen oder auch Datenworte aus dem Speicher zu laden oder zu speichern. Ebenfalls enthalten sind Befehle, um die Programmabfolge zu beeinflussen.

Bei der RISC-V ISA handelt es sich, wie der Name schon verrät, um eine RISC (Reduced Instruction Set Computer) Architektur. Die Menge an Maschinenbefehlen für einen voll funktionsfähigen Prozessor (oder in unserem Fall Emulator) beträgt lediglich 37 Befehle.

Alle Befehle haben eine feste Länge von 32 Bit, es werden daher immer für einen Befehl 4 Byte (ein Byte sind 8 Bit) aus dem Programmspeicher geladen. Der Instruktions- und Datenspeicherzugriff erfolgt aus Sicht des Prozessors immer wahlfrei, d. h., dass der Prozessor eine Speicheradresse generiert und den Wert an der Adresse entweder liest oder schreibt. Der Instruktionsspeicher wird in unserem Fall ausschließlich gelesen. Die Adresse für die nächste Instruktion, die gelesen werden soll, ist in dem Progammzähler Register (Program Counter PC) hinterlegt.

RISC Architekturen sind durch ihre Eigenschaft als Load/Store-Architektur definiert. Eine Load/Store-Architektur besitzt für die Datenspeicherzugriffe ausschließlich spezielle Lade- und Speicherbefehle, die Daten von und in den Speicher transferieren. Die Operanden für eine Rechenoperation kommen ausschließlich aus den Registern oder sind festkodiert im Befehl, diese werden immediate Operanden genannt.
Die RISC-V Architektur besitzt 32 Register, die in dem Registerfile zusammen organisiert sind. Der Wert in Register 0 ist und bleibt immer 0, er verändert sich nicht.

Nachdem ein Befehl aus dem Instruktionsspeicher geladen wurde, muss er dekodiert werden. In der Basis-ISA RV32I gibt es sechs Befehlsformate: R, I, S, B, U und J siehe Tabelle 1.

- Das **R Befehlsformat** ist das Register/Register Format. Das bedeutet, dass zwei Operanden aus dem Registerfile geladen werden und das Ergebnis einer ALU-Operation in das Registerfile gespeichert wird.
- Bei dem **I Befehlsformat** wird der zweite Operand aus der Instruktion extrahiert und das Ergebnis, wie bei dem R Befehlsformat, in das Registerfile geschrieben. Instruktionen, die dafür da sind, aus dem Datenspeicher zu lesen, verwenden ebenfalls das I Befehlsformat.
- Instruktionen, die einen Registerwert in den Datenspeicher schreiben, verwenden das **S Befehlsformat**.
- Die bedingten Sprünge (branches) verwenden das **B Befehlsformat** und die "Jump and Link"Instruktion verwendet das **J Befehlsformat**.
- Die beiden Instruktionen LUI (Load Upper Immediate) und AUIPC (Add Uper Immediate to PC) verwenden das **U Befehlsformat**.

In Tabelle 1 sind sämtliche Instruktionen aufgelistet, die Ihr Emulator ausführen muss. Die Funktionen der Befehle entnehmen Sie Tabelle 2.

Wenn auf die Adresse 0x5000 ein Byte geschrieben wird (Maschinenbefehl „Store Byte“ SB), soll der Emulator ein Zeichen auf dem Terminal ausgeben. Dieses Vorgehen ist als „speicherabgebildete Ein- und Ausgabe“ oder „Memory Mapped IO“ bekannt. Bei diesem Ein-/Ausgabeschema werden Teile des Adressraums den verschiedenen Ein-/Ausgabegeräten zugeordnet und Lese- und Schreiboperationen für diese Adressen als Befehle für das jeweilige Ein-/Ausgabe-Gerät interpretiert. In unserem Fall soll nur geschrieben werden und auch nur ein Byte auf die Standardausgabe ausgegeben werden.

2 Dateien

- In der main.c Datei sind die Funktionen für das Laden der Instruktions- und Datenspeicher enthalten. In diese main.c Datei sollen Sie Ihren kompletten Source Code für den RISC-V Emulator hineinschreiben. Für die Abgabe Ihres Projekts ist ausschließlich main.c relevant. Dementsprechend ist es ausnahmslos untersagt, Ihren Quelltext auf mehrere C-Dateien zu verteilen.

- In der Datei instruction_mem.bin ist ein Programm für den Emulator enthalten.
- Die Datei data_mem.bin enthält Daten für das Programm.

3 Vorgehen

Das Programm für den Emulator und der Inhalt des Datenspeichers sind in zwei separaten Binärdateien hinterlegt (Harvard-Architektur). Die einzelnen Schritte für eine Emulation des Prozessors sind wie folgt:

1. Es wird ein 32 Bit Befehl aus dem Programmspeicher geladen.
2. Dieser Befehl wird mithilfe von Bitmustern und Schiebeoperationen decodiert.
3. Der Befehl wird ausgeführt. Je nachdem um welche Art von Befehl es sich handelt, wird entweder das Registerfile aktualisiert, von dem Datenspeicher gelesen oder geschrieben oder ein Sprung vorbereitet bzw. in das Programmzählerregister (PC) geschrieben.
4. Der Emulationsprozess beginnt von vorne mit 1).

Datenstruktur:
Als grundlegende Datenstruktur für Ihren Emulator verwenden Sie das `struct CPU`:

```
typedef struct {
    // size of data memory
    size_t data_mem_size;
    // the register file
    uint32_t regfile_[32];
    // the program counter
    uint32_t pc;
    // the instruction memory
    uint8_t* inst_mem;
    // the data memory
    uint8_t* data_mem;
} CPU;
```

In der Datei `main.c` gibt es bereits einige Hilfsfunktionen, welche zur Initialisierung einer `CPU`-Instanz verwendet werden. Um das Programmierprojekt erfolgreich abzuschließen, müssen Sie folgende Funktion implementieren:

```
void
CPU_execute(CPU* cpu) {
    // TODO
}
```

Decodierung: Die Programmiersprache C besitzt keine direkte Möglichkeit einzelne Bits zu verändern, es gibt aber einen Trick, die eine Manipulation ermöglicht. Damit die relevanten Bits richtig interpretieren werden, müssen diese an die richtige Stelle verschoben werden. Dafür ist in der Programmiersprache C der Schiebeoperator „>>“

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

Tabelle 1: RV32I Instruktionsformate und Befehle (`https://github.com/riscv/riscv-isa-manual`).

Maschinen-befehl	Beschreibung der Aktion
lui	register_file[rd] = imm; pc + 4;
auipc	register_file[rd] = pc + imm; pc + 4;
jal	register_file[rd] = pc + 4; pc = pc + (int32_t)imm;
jalr	register_file[rd] = pc + 4; pc = (register_file[r1]) + ((int32_t)imm);
beq	if (register_file[r1] == register_file[r2]) pc = pc + (int32_t)imm; else pc + 4;
bne	if (register_file[r1] != register_file[r2]) pc = pc + (int32_t)imm; else pc + 4;
blt	if (register_file[r1] < register_file[r2]) pc = pc + (int32_t)imm; else pc + 4;
bge	if (register_file[r1] >= register_file[r2]) pc = pc + (int32_t)imm; else pc + 4;
bltu	if (register_file[r1] < register_file[r2]) pc = pc + (int32_t)imm; else pc + 4;
bgeu	if (register_file[r1] >= register_file[r2]) pc = pc + (int32_t)imm; else pc + 4;
lb	register_file[rd] = data_mem[register_file[r1] + imm]; pc + 4; *
lh	register_file[rd] = *(uint16_t*)(register_file[r1] + imm + data_mem); pc + 4; *
lw	register_file[rd] = *(uint32_t*)(register_file[r1] + imm + data_mem);
lbu	register_file[rd] = data_mem[register_file[r1] + imm];
lhu	register_file[rd] = *(uint16_t*)(register_file[r1] + imm + data_mem);
sb	data_mem[register_file[r1] + (int32_t)imm] = (uint8_t)register_file[r2];
sh	*(uint16_t*)(data_mem + register_file[r1] + imm) = (uint16_t)register_file[r2]; pc + 4;
sw	*(uint32_t*)(data_mem + register_file[r1] + imm) = (uint32_t)register_file[r2]; pc + 4;
addi	register_file[rd] = register_file[r1] + imm; pc + 4;
slti	register_file[rd] = register_file[r1] < imm; pc + 4; *
sltiu	register_file[rd] = register_file[r1] < imm; pc + 4; *
xori	register_file[rd] = register_file[r1] xor imm; pc + 4;
ori	register_file[rd] = register_file[r1] | imm; pc + 4;
andi	register_file[rd] = register_file[r1] & imm; pc + 4;
slli	register_file[rd] = register_file[r1] << imm; pc + 4;
srli	register_file[rd] = register_file[r1] >> imm; pc + 4;
srai	register_file[rd] = register_file[r1] >> imm; pc + 4;*
add	register_file[rd] = register_file[r1] + register_file[r2]; pc + 4;
sub	register_file[rd] = register_file[r1] - register_file[r2]; pc + 4;
sll	register_file[rd] = register_file[r1] << register_file[r2]; pc +4;
slt	register_file[rd] = (int32_t)register_file[r1] < (int32_t)register_file[r2]
sltu	register_file[rd] = register_file[r1] < register_file[r2]
xor	register_file[rd] = register_file[r1] - register_file[r2]; pc + 4;
srl	register_file[rd] = register_file[r1] >> register_file[r2]; pc +4;
sra	register_file[rd] = (int32_t)register_file[r1] >> register_file[r2]; pc +4;
or	register_file[rd] = register_file[r1] | register_file[r2]; pc + 4;
and	register_file[rd] = register_file[r1] & register_file[r2]; pc + 4;

Tabelle 2: RV32I Befehle und ihre Aktionen. Bei den Maschinenbefehle die mit * gekennzeichnet sind, beachten Sie bitte das Vorzeichen!

vorgesehen. Um unrelevante Bits einer Variablen zu entfernen, kann die Variable mit einem Bitmuster „&“ verknüpft werden. Auf diese Weise können beispielsweise die Adressen der Quell- und Zielregister oder der Immediate Wert aus der Instruktion extrahiert werden. Dazu an dieser Stelle ein Beispiel:
Sie möchten die Adresse $rs1$ für den Quelloperand aus einer R-Type Instruktion extrahieren.

```
uint8_t rs1;
...
rs1 = (0x1F) & (uint_8_t)(*instruction >> 15));
...
```

Was passiert konkret bei der Extraktion von $rs1$ bei dem Assemblerbefehl?

```
sub x11, x11, x12?
```

Der entsprechende Maschinenbefehl in hexadezimaler Darstellung ist 0x40C585B3. Zunächst schieben wir 15 Stellen nach rechts. Das Resultat der Schiebeoperation ist $0x818B$. Wenn $0x818B$ mit dem Bitmuster $0x1F$ & verknüpft wird, bekommen wir als Ergebnis $0xB$ in Dezimaldarstellung 11. Die Adresse $rs1$ für das Registerfile ist also 11.

Ausführung: Nachdem der Emulator die einzelnen Bitfelder extrahiert hat, wird der entsprechende Maschinenbefehl ausgeführt. Für die Entscheidung, welcher Maschinenbefehl ausgeführt werden soll, bietet es sich an eine switch Anweisung auf das OPCODE Feld durchzuführen. Was konkret jeder der RISC-V32 Maschinenbefehle ausführt, ist in der Tabelle 2 aufgelistet. Führen Sie 140.000 Befehle aus. Die Ausgabe des Terminals sollte dann wie in Abbildung 1 aussehen.

4 Programmausführung auf dem Emulator

Auf der Moodle-Seite finden Sie zwei übersetzte RISC-V Programme, die Sie auf Ihrem Emulator ausführen können. Das erste Programm führt unterschiedliche arithmetische Operationen durch und gibt das Resultat auf der Standardausgabe aus. Das zweite Programm rechnet die Primzahlen von 0 bis 1999 aus und gibt sie ebenfalls auf der Standardausgabe aus. Der Aufruf des Emulators erfolgt im Terminal durch:

```
./hu-risc-av_emu instruction_mem.bin data_mem.bin
```

Sie müssen entweder den Pfad zu den Dateien instruction_mem.bin und data_mem.bin beim Aufruf mit angeben oder Sie kopieren die beiden Dateien in das Verzeichnis, in das Ihr Emulator übersetzt wurde.

Auf der Virtuellen Maschine für das Schaltkreispraktikum ist sämtliche Software installiert, um RISC-V Programme zu übersetzen. Sie können also auch selbstständig Programme erstellen, indem Sie das Beispielprojekt anpassen und auf der Virtuellen Maschine

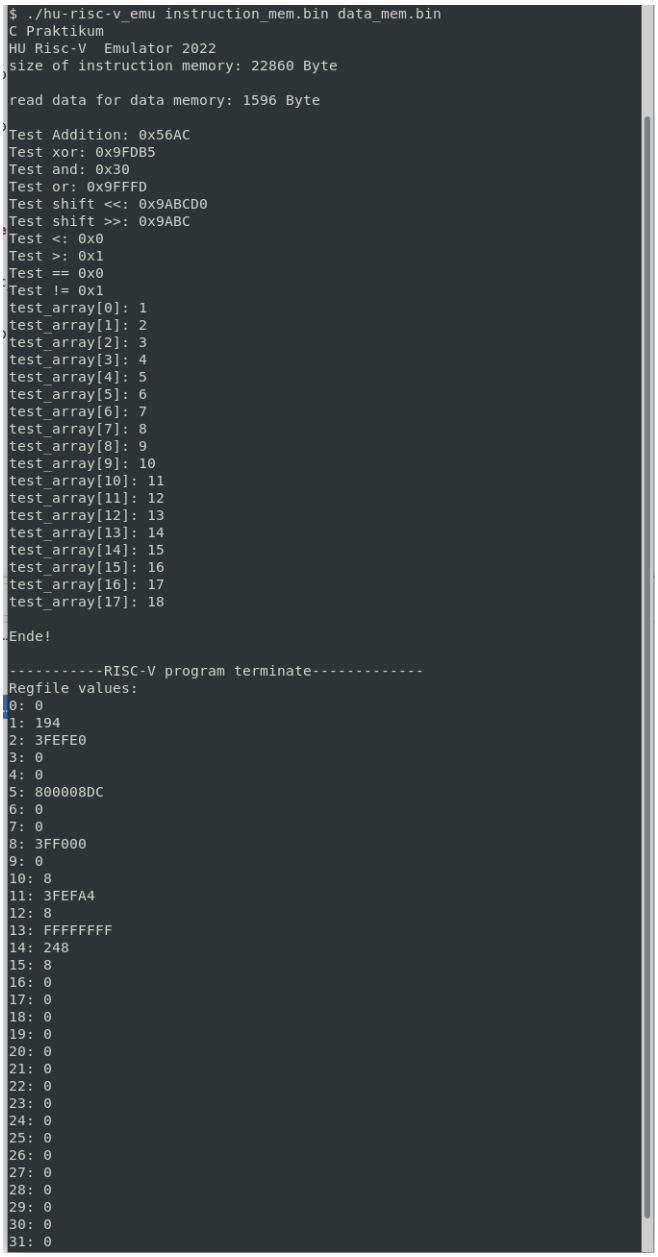

Abbildung 1: Standardausgabe des ersten RISC-V Programms

innerhalb des Projektordners den Befehl make eingeben. Nach der Ausführung von make werden die Dateien instruction_mem.bin und data_mem.bin erstellt, die Sie mit ihrem Emulator ausführen können.

5 Abgabe

main.c

Der komplette RISC-V Emulator soll in der main.c Datei von Ihnen programmiert werden. Die Verwendung weiterer Quelldateien in Abgaben ist nicht zulässig und steht nicht zur Diskussion. Weiterhin dürfen ausschließlich die in der Aufgabenstellung bereits inkludierten Header verwendet werden, die Verwendung weiterer Header ist nicht zulässig.

Das Programm soll mit dem Befehl

```
gcc main.c -o hu_risc-v_emu -std=c11
```

übersetzt werden können. Als Referenzsystem dient `gruenau4.informatik.hu-berlin.de`, auf diesem Rechner muss Ihr Code funktionieren. Abgaben, die dort nicht kompilieren oder dort nicht korrekt ausführt werden können, werden als nicht bestanden gewertet.

Die Aufgabe gilt als bestanden, wenn zum einen Ihr Emulatorprogramm sämtiche in der Tabelle 2 aufgelisteten Maschinenbefehle ausführen kann. Wir werden Ihren Emulator mit unterschiedlichen C-Programmen testen und empfehlen Ihnen, dies ebenfalls zu machen.

6 Zeitplan

Sie können sofort mit der Bearbeitung der Aufgabe beginnen. Bis zur finalen Abgabefrist haben Sie beliebig viele Abgabeversuche, eine erste Abgabe ist ab sofort möglich. Wir bemühen uns, Ihnen nach jedem Abgabeversuch kurzfristig Rückmeldung zu geben, ob Ihre Lösung korrekt ist oder, falls nicht, inwiefern sich Ihr Programm falsch verhält. **Nach der finalen Abgabefrist werden keinerlei weitere Abgaben mehr berücksichtigt.**

Wir ermutigen Sie ausdrücklich, so früh wie möglich mit der Bearbeitung der Aufgabe zu beginnen und einen ersten sinnvollen Lösungsversuch bis Ende Juli abzugeben. Je nach Aufkommen von Abgaben können wir wenige Wochen vor der finalen Deadline nicht mehr garantieren, kurzfristig individuelles Feedback zu geben, das Ihnen auch wirklich weiter hilft. Da eine vollständig korrekte Lösung für das Bestehen erforderlich ist, sind erfahrungsgemäß fast immer mehrere Abgaben notwendig[1].

Wenn Sie eine korrekte Lösung eingereicht haben, müssen Sie uns für die Erbringung der Modulteilleistung zusätzlich noch im Rahmen eines kurzen persönlichen Abtestats

[1] Die Deadlines sind ernst gemeint. Von einer Abgabe ein paar Stunden, Minuten oder Sekunden vor der finalen Abgabefrist raten wir dringend ab. Technische Probleme bei der Abgabe sind kein Grund für eine Ausnahme von der Frist.

Ihre Lösung erläutern und Fragen dazu beantworten. Die Abtestate werden online über Zoom stattfinden.

Datum	Beschreibung
30.07.2022	Empfohlene Deadline für eine erste Abgabe; bei Abgabe bis zu diesem Termin können wir im Falle von Problemen mit der Abgabe gewährleisten, dass nach einer ersten Rückmeldung noch Zeit zur Nachbesserung bis zur endgültigen Deadline bleibt.
01.09.2022 (23:59 Uhr MESZ)	Spätestmögliche Deadline für die Abgabe der Endversion, die alle Kriterien vollständig erfüllen muss.
13.09.2022 – 15.09.2022	In diesem Zeitraum finden die mündlichen Testate in Einzelsitzungen statt. Eine frühere Abnahme ist nach individueller Absprache möglich.

Viel Erfolg!