Computerarchitectuur 2016
Inleveropdracht 2: Simple RISC-V 64 emulator

Gesuggereerde Deadline: zondag 6 november 2016

Het doel van de tweede inleveropdracht is om een eenvoudige emulator te schrijven welke eenvoudige RISC-V programma's kan uitvoeren. We kiezen voor RISC-V vanwege de eenvoud en we zullen ons beperken tot een klein aantal instructies. Op de website van het vak is een startpunt te downloaden waarin een aantal zaken al is geïmplementeerd: het laden van programma's in ELF formaat, eenvoudige emulatie van een geheugenbus en geheugen, register file en simpele "devices" om karakters als uitvoer te genereren en het systeem stop te zetten (halt).

We beperken ons tot het uitvoeren van instructies. We simuleren niet de exacte karakteristieken van DDR geheugen en we modelleren en simuleren geen cache systeem. We gaan ervan uit dat we binnen één klokcyclus ook een memory load/store kunnen uitvoeren. Met een verdere uitbreiding van de emulator zouden dit soort zaken uiteraard wel in detail kunnen worden nagebootst.

Op de website is ook een collectie van testprogramma's te vinden. Deze hebben een oplopende complexiteit: `add-riscv.bin`, `hello.bin`, `sumdemo.bin`, `comp.bin`, `matvec.bin`, `matvecu.bin`. Gedurende de ontwikkeling van de emulator zal het nodig zijn om eenvoudige testprogramma's te schrijven om individuele instructies te testen. We noemen dit "micro-programs" en je kunt `add-riscv.s` als voorbeeld hiervoor gebruiken.

Achtergrondinformatie

Algemene theorie over Instruction Set Architectures kan worden gevonden in Appendix A van het tekstboek (Hennessy en Patterson) en het hoorcollege over Appendix A. In Appendix C is de "klassieke RISC" pipeline omschreven met de vijf stappen voor het uitvoeren van een instructie, welke we in de opdracht zullen volgen. Hoofdstuk C3 omschrijft de implementatie van een pipeline voor een eenvoudige MIPS processor.

Gedetailleerde informatie over de RISC-V instructieset kan worden gevonden in de architecture reference manual: `https://content.riscv.org/wp-content/uploads/2016/06/riscv-spec-v2.1.pdf`.

Aanpak

Belangrijk: om het startpunt te kunnen compileren heb je een moderne C++-compiler nodig (tenminste g++ 5.x of Clang 3.5). Als je op de computers van de Universiteit werkt (inclusief huisuil): zorg dat je in het `ca2016` environment zit:

```
source /vol/share/groups/liacs/scratch/ca2016/ca2016.bashrc
```

Om tot een werkende emulator te komen moeten de volgende delen worden aangepast:

1. `inst-decoder.cc`: hier moet de instruction decoder in terecht komen, waar je aan bent begonnen in werkcollege 4. De functie `decode_instruction` uit werkcollege 4 vind je hier terug als de methode `InstructionDecoder::decodeInstruction`.

2. `inst-decoder.h`: hier vul je `struct DecodedInstruction` in. Tevens kun je methoden toevoegen aan `InstructionDecoder` die je later nodig hebt om in `class Processor` alles aan elkaar te knopen.

3. `inst-formatter.cc`: in dit bestand moet de code terecht komen om de instructies in tekstuele vorm op het scherm te zetten. De functie `show_decoded_instruction` van werkcollege 4 is hier de functie `operator<<(std::ostream &os, const DecodedInstruction &decoded)`.

4. `alu.cc`, `alu.h`: in deze bestanden wordt het ALU component geïmplementeerd. Hier moet alle logica terecht komen die nodig is om de berekening voor een instructie te kunnen uitvoeren. Er zijn methoden om de operanden te zetten en het resultaat te lezen. Nog toe te voegen zijn methoden om de uit te voeren operatie op de ALU in te stellen (bijv. door opcode en/of functiecode door te geven). Als op de ALU `execute()` wordt aangeroepen, voert de ALU de gezette operatie uit op de gezette operanden. Hierna kan het resultaat worden uitgelezen.

5. `processor.cc`, `processor.h`: uiteindelijk moet in `Processor` alles aan elkaar worden geknoopt. In de header file valt te zien dat `Processor` voor elk component in het systeem een member-variabele bevat. De methoden voor de verschillende stappen moeten in het `.cc`-bestand worden geïmplementeerd: `instructionFetch`, `instructionDecode`, enz. Bijvoorbeeld voor "instruction decode" moet de gedecodeerde instructie worden geïnterpreteerd en de juiste operanden en codes moeten op de ALU worden gezet. Bij "execute" wordt op de ALU `execute()` aangeroepen om een operatie uit te voeren. Enzovoort.

We gaan ervan uit dat elk van de vijf stappen één klokcyclus nodig heeft. Het uitvoeren van een reg-reg ALU instructie vereist alle vijf stappen en kost dus vijf klokcycli.

Merk tevens op dat de emulator op deze manier steeds aan één instructie tegelijk werkt, dus non-pipelined! In eerste instantie werk je aan een non-pipelined executie van instructies.

Het verdient de aanbeveling om de instructies één-voor-één, of in kleine groepjes van soortgelijke instructies, te implementeren. Dus voor een bepaalde instructie schrijf je eerst decode, dan formatting, ALU en uiteindelijk wat nog nodig is in `class Processor`. Vervolgens kun je deze individuele instructie(s) testen. Hiervoor is het handig om een "micro-program" te schrijven. Je kunt `add-riscv.s` als voorbeeld gebruiken. Het is mogelijk om een "micro-program" te compileren dat maar uit 1 instructie bestaat. Je kunt dit programma uitvoeren met *vooraf* geïnitialiseerde registers, zodat je kunt nagaan of de instructie werkt. Voorbeeld:

```
./rv64-emu -r r1=10 -r r2=40 ../rv64-examples/add-riscv.bin
```

Wanneer de emulator stopt, worden alle huidige waarden van de registers afgedrukt.

Opgaven

1. Maak allereerst de benodigde aanpassingen zodat de emulator de `addw` instructie kan uitvoeren (non-pipelined). Implementeer instruction fetch en laat de PC steeds met 4 bytes toenemen. Hierdoor zal de emulator vanzelf een keer voorbij het einde van het geheugen proberen te lezen, waardoor het wordt afgesloten met een abnormal program termination. (Het is normaal dat de emulator uit het startpunt zonder aanpassingen in een oneindige loop terechtkomt).

   Gebruik het programma `add-riscv.bin` om de implementatie van de instructie te testen. Maak hierbij gebruik van de mogelijkheid de registers via de command-line argumenten te initialiseren. Kloppen de resultaten wanneer de emulator stopt met werken?

2. Inventariseer welke instructies moeten worden geïmplementeerd om `hello.bin` uit te voeren.

3. Implementeer deze instructies één-voor-één (weer non-pipelined). Schrijf micro-programma's om te testen wanneer nodig. Merk overigens op dat `hello.bin` aan het einde een store-operatie uitvoert naar een specifiek geheugenadres waaraan de system control module is gekoppeld. Deze zorgt ervoor dat de emulator netjes "stopt" en wordt afgesloten. Bij een juiste uitvoering van `hello.bin` zal er dus geen "abnormal program termination" plaatsvinden.

4. Ga op dezelfde manier verder met de complexere programma's.

5. Omschrijf in een kort verslag hoe je in `class Processor` de verschillende componenten aanstuurt (en aan elkaar knoopt) en welke keuzes je hierbij hebt gemaakt.

6. Omschrijf in het verslag ook hoe je de emulator hebt getest. Heb je hier extra programma's voor geschreven, zo ja wat voor? Hoe heb je dit aangepakt?

7. Omschrijf in het verslag duidelijk welke onderdelen wel en niet werken.

8. Denk na hoe pipelining in deze emulator zou moeten worden geïmplementeerd. Geef in het verslag een omschrijving, eigenlijk een plan van aanpak, van de aanpassingen die nodig zijn aan de emulator om ervoor te zorgen dat deze "pipelined" de instructies kan uitvoeren en dus aan meerdere instructies tegelijkertijd kan werken. Wat is hier voor nodig? Hoe pak je dit aan? Denk aan pipeline registers, store forwarding, hazards, interlock.

9. *(Optioneel)* Implementeer echte pipelining in de emulator. Laat zien dat door pipelining de CPI voor de verschillende testprogramma's wel of niet omlaag gaat.

Inleveren

Er mag worden gewerkt in duo's. Het volgende moet worden ingeleverd:

- Source code van de aangepaste emulator. *Géén* binaries of object files. Gebruik `make clean`!
- Het verslag, gewoon in tekstformaat: `verslag.txt`.

Zorg ervoor dat alle bestanden die worden ingeleverd (source code, verslag, enz.) zijn voorzien van naam en studentnummer! Plaats alle bestanden om in te leveren in een aparte directory (bijv., `opdracht2`) en maak een "gzipped tar" bestand (het is prima als source code en verslag in dezelfde `tar.gz` terechtkomen):

```
tar -czvf opdracht2-sXXXXXXX-sYYYYYYY.tar.gz opdracht2/
```

Vul op de plek van `XXXXXXX` en `YYYYYYY` de bijbehorende studentnummers in. De inzendingen kunnen worden verzonden per e-mail naar `ca2016 (at) handin.liacs.nl` met als onderwerp "CA Opdracht 2". Vermeld in de e-mail ook namen en studentnummers.