A2: Simulering af x86prime - x86_64 delmængde med udvidelser

Computer Systems 2018
Department of Computer Science
University of Copenhagen

Finn Schiermer Andersen

Due: Søndag, 7. Oktober, 10:00
Version 1

Dette er den tredje afleveringsopgave i Computersystemer og den første som dækker Maskinarkitektur. Som tidligere, opfordrer vi til par-programmering og anbefaler derfor at lave grupper at 2 til 3 studerende. Grupper må ikke være større end 3 studerende og vi advarer mod at arbejde alene.

Afleveringer vil blive bedømt med op til 4 point. Du må opnå mindst halvdelen af mulige point over kurset for at blive indstillet til eksamen. Du kan finde yderligere detaljer under siden "Course description" på Absalon. Denne aflevering høre under Maskinarkitektur. Det er *ikke* muligt at genaflevere afleveringer.

Introduction

> *There is only one mistake that can be made in a computer design that is difficult to recover from – not providing enough address bits for memory addressing and memory management.*
>
> — Gordon Bell, *Computer Structures: What Have We Learned from the PDP11?* (1976)

Denne afleveringsopgave består i at færdiggøre en simulator for et mindre instruktionssæt, defineret over assembler sproget vi kalder *x86prime*. X86prime er i overvejende grad en delmængde af x86_64 assembler sproget, men der er også enkelte instruktioner i x86prime som ikke indgår i x86_64.

Til løsning af opgaven udleverer vi en halvfærdig simulator som kun kan udføre en enkelt instruktion. Det er denne simulator der skal færdiggøres og testes; dertil skal arbejdet dokumenteres i jeres rapport.

1 Assember sproget x86prime

Som nævnt er x86prime defineret som en delmængde at x86_64, som I kender fra BOH. Grunden til at vi i denne opgave ikke benytter x86_64, er at assember sproget (og det tilhørende instruktionssæt) er alt for stort og har en for kompleks semantik til at I kan forstå og bygge simulator over opgaveforløbet. x86prime er dog valgt så det dækker mange af mest brugte instruktioner

fra x86_64 og uvidelserne har simple program transformationer til/fra x86_64. Forståelse af x86_64 giver derfor en forståelse af x86prime og vice versa.

Til x86prime bruger vi følgende delmængde af x86_64 instruktionerne:

- `MOVQ %ra, %rb`: kopiering fra et register til et andet,
- `MOVQ $imm, %rb`: initialisering af register,
- `MOVQ $imm(%rb), %ra`: læsning fra lageret,
- `MOVQ %ra, $Imm(%rb)`: skrivning til lageret,
- `ADDQ/SUBQ/ANDQ/ORQ/XORQ/MULQ %ra, %rb`: aritmetik på registre,
- `ADDQ/SUBQ/ANDQ/ORQ/XORQ/MULQ $imm, %rb`: aritmetik med heltal,
- `JMP target`: ubetinget hop,
- `LEAQ $imm(%rs,%rz,$scale), %ra`.

Læg mærke til at `MOVQ` instruktioner *kun* giver mulighed for ét addresseberegnings register, samt *ikke* har mulighed for at skallerer det ene.

Dertil udvider vi med følgende instruktioner som ikke er originale x86 instruktioner:

- `CALL target, %ra`: underprogramkald som gemmer returadressen i `%ra`,
- `RET %ra`: retur fra underprogramkald som læser returadressen fra `%ra`,
- `CBcc %ra, %rb, target`: hop hvis relationen `%ra` *cc* `%rb` er opfyldt,
- `CBcc $imm, %rb, target`: hop hvis relationen `$imm` *cc* `%rb` er opfyldt.

Forskellen i ovelstående til x86_64 er at kald til og returnering fra procedurer håndteres ved at gemme/hente returaddressen fra et register og *ikke* kaldstakken; som generel konvention vil vores oversættes af C og x86_64 kode benytte `%r15` til dette. Dertil benytter x86prime ikke betingelsesflag til at foretage betingede hop, men sammensættet både test og hop i instruktionen `CBcc`.

En anden ændring er at x86prime benytter en anden indkodning af ordrer end x86_64; altså en anden oversætte fra assember sprog til maskinkode. Dette er igen for at gøre opgaven nemmere for jer. Instruktionerne og indkodningen er nærmere beskrevet i filen `encoding.txt`, som ligger sammen med den udleverede kildetekst, og vist i Bilag A.

2 Hjælpeværktøjer

Til hjælp med at arbejde med med x86 prime, udleverer vi et program, `x86prime`, der kan fungere både som assembler, reference-simulator og kryds-oversætter fra x86_86 til x86prime som det genereres af gcc:

- Programmet kan indkode x86prime assembler programmer til et hexadecimalt format, som kan indlæses af den udleverede halvfærdige simulator.

- Programmet kan indlæse x86 assembler programmer genereret med gcc og oversætte dem til x86prime assembler. Denne funktionalitet er begrænset og muligvis ikke fejlfri.
- Programmet kan simulere x86prime programmer og generere en "sporingsfil". Denne fil kan indlæses af den udleverede halvfærdige simulator, hvor den bruges til at holde den ufærdige simulators opførsel op imod reference-simulatorens.

Du finder kildeteksten på GitHub her:

`https://github.com/finnschiermer/x86prime`

Du er velkommen til selv at oversætte værktøjet lokalt på din maskine, men eventuelle problemer du oplever må du selv løse[1]. Vi frigiver et billede til en virtual maskine hvor værktøjet er pre-installeret og har de nødvendige biblioteker til oversættelse.

3 Kildetekst til simulator

Den halvfærdige simulator som kun kan køre `RET %r15` instruktionen, finder du i en zip-fil der distribueres sammen med denne opgavetekst.

Til hjælp findes i Figur 1 et forslag til en datavej der kan bringes til at udføre alle de ønskede instruktioner. Det er ikke et krav at simulatoren følger disse dataveje, men variationer bør argumenteres for i rapporten.

Den halvfærdige simulator består af følgende filer:

- `main.c` - Hovedprogram. Du behøver kun at rette i denne fil for at løse opgaven.
- `wires.h`, `wires.c` - Funktioner der implementerer til forbindelser/ledninger på en processor.
- `arithmetic.h`, `arithmetic.c` - Funktioner der bruges til grundlæggende aritmetik og digital logik.
- `compute.h`, `compute.c` - En regneenhed specifikt designed til x86prime.
- `memory.h`, `memory.c` - Lageret (Main memory).
- `registers.h`, `registers.c` - Registerfilen.
- `ip_reg.h`, `ip_reg.c` - IP-registeret (Program Counter).

[1] Brug forum for kurset til at bede om hjælp til at løse problemer. Der er en sandsynlighed for at en medstuderende kender til det.

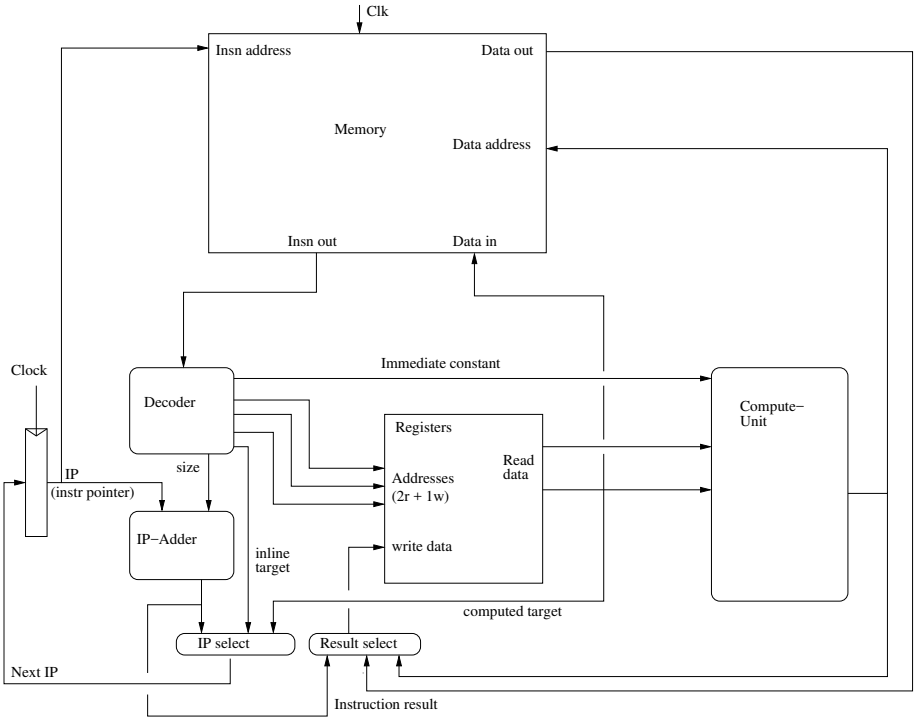

Figure 1: Diagram som viser datavejen for en simpel enkelt-cyklus processor.

3.1 Byggeklodser

Den halvfærdige simulator skal færdiggøres og det skal ske ved at bruge de funktioner og typer der er udleveret i filerne. Det skyldes at I skal bygge en simulator som emmulerer en faktisk elektronisk processor, hvilket igen sætter begrænsninger på hvordan den skal implementeres.

Store dele at C har ikke nogen direkte implementation som logiske kredsløb og når højre-niveau sprog bliver benyttet til design af kredsløb skal programmet igennem en avanceret proces kaldet logisk syntese. Det at vi ikke til rådighed.

Der er nogle ting I gerne må bruge:

- I må gerne bruge de udleverede funktioner. Disse er "byggeklodser" til at lave simuleringen med.
- I må gerne bruge variabler af typen `bool` og `val`. Disse fungerer som "ledninger" i simulationen.
- I må gerne bruge Boolsk logik (`||`, `&&` ovs.). Disse fungerer som logiske porte.

Og nogle ting I *ikke* må bruge:

- I må ikke arbejde med 64-bit værdier direkte. I skal bruge typen `val`, samt de udleverede funktioner der arbejde værdier af denne type.

- I må ikke bruge conditionals (`if`, `else`, `?`-operatoren). Det kan strøm ikke gøre.
- I må ikke bruge løkker (`for`, `while`) udover hovedløkken der allerede er skrevet.
- I må ikke sætte en variabels værdi mere end én gang. En ledning kan kun holde en strøm. Dette kender I fra implementationer i F#

De ting I ville gøre med conditionals kan typisk gøres med funktioner fra `wires.h`.

For at løse opgaven er det tilstrækkeligt at ændre/tilføje til koden i `main()` i filen `main.c`.

4 Bedømmelse og vejledning

De forskellige dele af afleveringen bliver vurderet efter følgende:

- 15% for en løsning der håndterer alle `MOVQ` instruktioner, inklusiv test af disse.
- 15% for en løsning der håndterer de aritmetisk/logiske og alle `LEAQ` instruktioner, inklusiv test af disse.
- 15% for en løsning der håndterer `CALL`, `JMP` og `CBcc`, inklusiv test af disse.
- 15% for generel test af de implementerede dele af simulatores inklusiv tydeliggørelse af evt. mangler og afvigelser.
- 40% Rapport som dokumenterer jeres implementation.
  - Beskriv hvordan jeres kode oversættes og hvordan jeres tests skal køre for at genskabe jeres resultater.
  - Diskuter alle ikke-trivielle dele af jeres implementation og design beslutninger.
  - Beskriv alle tvetydige formuleringer, som I kan have fundet i opgave teksten.

Det er forventeligt at rapporten er omkring 5 sider og den må ikke overskrive 7 sider.

For at få point skal man kunne dokumentere at opgaven er løst korrekt. Det gøres ved at udarbejde og køre testprogrammer og bekræfte at den udarbejdede løsning laver de samme ændringer til lager og registre som referencesimulatoren. Overvej både test for specifikke instruktioner og sammenhænget mellem flere instruktioner.

Vi anbefaler at man løser opgaven i tre faser svarende til de tre første punkter ovenfor, da de er rangeret efter sværhedsgrad. På den måde ved man rimelig sikkert, hvornår et point er i hus. Det er så trist at få alt til at virke, og så ikke have tid til at teste det.

Sammen med jeres rapport, `report.pdf`, skal I aflevere en `src.zip` som indeholder jeres udviklede simulator og test programmer, samt en `group.txt`. `group.txt` skal ASCII/UTF8 formateret liste KU-id'er fra alle medlemmer i gruppen; et id pr. line, ved brug af følgende tegnsæt:

$$\{\texttt{0x0A}\} \cup \{\texttt{0x30}, \texttt{0x31}, \ldots, \texttt{0x39}\} \cup \{\texttt{0x61}, \texttt{0x62}, \ldots, \texttt{0x7A}\}$$

A encoding.txt

```
x86prime instruction encoding
-----------------------------

00000000 0000ssss                           ret s
0001aaaa ddddssss                           register/register arithmic: op s,d (see below)
00100001 ddddssss                           movq s,d
00110001 ddddssss                           movq (s),d
00111001 ddddssss                           movq d,(s)
-
0100cccc ddddssss pp...32...pp              cb<c> s,d,p (see below)
01001110 dddd0000 pp...32...pp              call p,d
01001111 00000000 pp...32...pp              jmp p
0101aaaa dddd0000 ii...32...ii              imm/register arithmetic: op i,d (see below)
01100100 dddd0000 ii...32...ii              movq $i,d
01110101 ddddssss ii...32...ii              movq i(s),d
01111101 ddddssss ii...32...ii              movq d,i(s)
-
10000001 ddddssss                           leaq (s),d
10010010 dddd0000 zzzzvvvv                  leaq (,z,(1<<v)),d
10010011 ddddssss zzzzvvvv                  leaq (s,z,(1<<v)),d
10100100 dddd0000 ii...32...ii              leaq i,d
10100101 ddddssss ii...32...ii              leaq i(s),d
10110110 dddd0000 zzzzvvvv ii...32...ii     leaq i(,z,(1<<v)),d
10110111 ddddssss zzzzvvvv ii...32...ii     leaq i(s,z,(1<<v)),d
-
1111cccc dddd0000 ii...32...ii pp...32...pp cb<c> $i,d,p (see below)

Explanations:

aaaa indicates the kind of arithmetic operation.
0000 add
0001 sub
0010 and
0011 or
0100 xor
0101 mul

d,s and z are registers.
0000 %rax     1000 %r8
0001 %rbc     1001 %r9
0010 %rcx     1010 %r10
0011 %rdx     1011 %r11
0100 %rbp     1100 %r12
0101 %rsi     1101 %r13
0110 %rdi     1110 %r14
0111 %rsp     1111 %r15

vvvv is a shift amount, only 0000,0001,0010 and 0011 are valid encodings.
When used in assembly, the values is given as scale factors 1,2,4 or 8.
```

```
ii...32...ii is a 32 bit signed immediate
pp...32...pp is a 32 bit target address

<c> is a condition mnemonic used in compare-and-branch. The compare-and-branch
instruction cb<c> is not part of the original x86 instruction set.

Example: cble %rdi,%rbp,target     = if %rdi <= %rbp then jump to target

Encoding
0000 e            1000 a
0001 ne           1001 ae
0010 <reserved>   1010 b
0011 <reserved>   1011 be
0100 l            1100 <reserved>
0101 le           1101 <reserved>
0110 g            1110 <reserved>
0111 ge           1111 <reserved>

call places the return address in %r15 instead of pushing it onto the stack.
ret returns to the address in %r15 instead of popping it from the stack.
```