A2: Simulering af x86 delmængde

Computer Systems 2017
Department of Computer Science
University of Copenhagen

Finn Schiermer Andersen

Due: Søndag, 1. Oktober, 23:59
Version 1 (September 19)

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 3 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)

Opgaven består i at færdiggøre en simulator for en delmængde af x86 assembler. Vi udleverer en halvfærdig simulator som kan udføre tre af de mest simple ordrer. Den skal færdiggøres, testes og testen skal dokumenteres.

1 x86 delmængde

Vi bruger følgende delmængde af x86 instruktionerne:

- `HALT` stopper simulationen
- `NOP` gør ingenting
- `MOVQ %ra,%rb` kopiering fra et register til et andet
- `MOVQ $I,%rb` initialisering af register
- `MOVQ D(%rb),%ra` læsning fra lageret
- `MOVQ ra,D(%rb)` skrivning til lageret

- `CMOVcc %ra,%rb` betinget tildeling
- `ADDQ %ra,%rb`, `SUBQ %ra,%rb`, `ANDQ %ra,%rb`, `XORQ %ra,%rb`, `CMPQ` aritmetik
- `Jcc target` betinget hop
- `CALL target` underprogramkald
- `RET` `return` fra underprogramkald
- `PUSH %ra` skub registerindhold på stakken
- `POP %ra` hent registerindhold fra stakken

Vi bruger en anden indkodning af ordrer end den rigtige x86 for at gøre opgaven nemmere for jer. Indkodningen er nærmere beskrevet i filen "encoding.txt" som ligger sammen med den udleverede kildetekst og vist i Bilag A.

Den halvfærdige simulator kan klare `HALT`, `NOP` og `MOVQ %ra,%rb`. Resten skal I selv lave.

2 Udleveret kildetekst

Vi udleverer en assembler og en reference simulator.

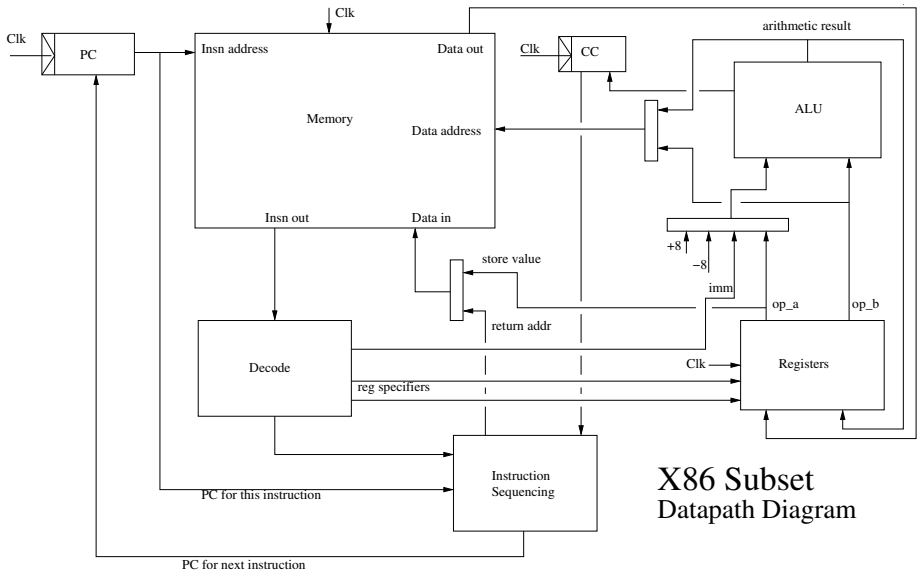
- Assembleren kan producere indkodede x86 ordrer som kan læses ind i lageret på den halvfærdige simulator og på reference simulatoren.
- Reference simulatoren kan producere "trace" filer. Disse filer beskriver de skrivninger til lager og registre som et program foretager under udførelsen. Trace-filerne kan læses af den udlevere halvfærdige simulator og bruges til at validere at opgaven løses korrekt.

Du finder kildeteksten på GitHub her:

- `https://github.com/kirkedal/compSys-e2017-sim`.

Den halvfærdige simulator (som skal færdiggøres i denne opgave) finder du i en zip-fil der distribueres sammen med denne opgavetekst.

Til hjælp er her et forslag til en datavej der kan bringes til at udføre alle de ønskede instruktioner:



Den halvfærdige simulator består af følgende filer:

- `main.c` - hovedprogram. Du behøver kun at rette i denne fil for at løse a2
- `wires.h`, `wires.c` - funktioner der svarer til forbindelser/ledninger
- `arithmetic.h`, `arithmetic.c` - funktioner til grundlæggende aritmetik og digital logik
- `alu.h`, `alu.c` - en ALU specifikt designed til vort x86 subset
- `memories.h`, `memories.c` - Lagerblokke. Registre.

3 Byggeklodser

Den halvfærdige simulator skal færdiggøres og det skal ske ved at bruge de funktioner og typer der er udleveret.

Det er *ikke* tilladt at arbejde direkte på indholdet i variable af typen "`val`". Disse variable må udelukkende manipuleres af de funktioner der allerede er givet i de udleverede `.h` filer.

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 at teste at den udleverede simulator virker korrekt for `HLT`, `NOP` og `movq %ra,%rb`

- 15% for en løsning der håndterer betinget tildeling (CMOVcc), betinget hop samt de aritmetiske instruktioner.
- 15% for en løsning der håndterer de øvrige `MOVQ` instruktioner.
- 15% for en løsning der håndterer `CALL`, `RET`, `PUSH` og `POP`.
- 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 3 til 5 sider og dem må ikke overskrive 5 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.

Vi anbefaler at man løser opgaven i fire faser svarende til de fire første punkter ovenfor. 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 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

```
X86 subset encoding

halt            0x00
nop             0x10
movq %a,%b      0x20 ab
cmovcc %a,%b    0x2v ab                         v!=0, see encoding of cc into v below
movq $i,%b      0x30 0b ii ii ii ii
movq %a,i(%b)   0x40 ab ii ii ii ii
movq i(%b),%a   0x50 ab ii ii ii ii
OPq %a,%b       0x6z ab                         see encoding of OP into z below
jcc t           0x7c ii ii ii ii                see encoding of cc into c below
call t          0x80 ii ii ii ii
ret             0x90
pushq %a        0xA0 a0
popq %a         0xB0 a0

Immediates for jcc and call are absolute addresses.
'i' is little endian (least significant byte first in instruction stream).

Encoding of v (for use in conditional move)
also encoding of cc (for use in conditional branch)

0 = always (jmp or movq)
1 = less or equal (jle, cmovle)
2 = less (jl, cmovl)
3 = equal (je, cmove)
4 = not equal (jne, cmovne)
5 = greater or equal (jge, cmovge)
6 = greater (jg, cmovg)

Encoding of z (operation in OPq)
0 = addq
1 = subq
2 = andq
3 = xorq
4 = cmpq
```