# A2: Simulering af x86prime - x86\_64 delmængde med udvidelser

Computer Systems 2019
Department of Computer Science
University of Copenhagen

Finn Schiermer Andersen

Due: Søndag, 6. 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 skal opnå mindst halvdelen af mulige point over kurset for at blive indstillet til eksamen, samt 40 % inden for hvert underemne. 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 tre forskellige instruktioner. Det er denne simulator der skal færdiggøres og testes; dertil skal arbejdet dokumenteres i jeres rapport.

Dertil skal der også løses en teoretisk opgave, som findes i Afsnit 4.

## 1 Assembler 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 de 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/IMULQ/SAR/SAL/SHR %ra, %rb: aritmetik på registre,
- ADDQ/SUBQ/ANDQ/ORQ/XORQ/MULQ/IMULQ/SAR/SAL/SHR \$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 skalere 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 ovenstående 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 oversættelse af C og x86\_64 kode benytte %r11 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 oversættelse fra assembler 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 programmerne, prasm, prun, primify, der fungere som hhv. assembler, reference-simulator og kryds-oversætter fra gcc generet x86\_86 til x86prime:

primify indlæser x86 assembler programmer genereret med gcc og oversætter dem til x86prime assembler. Denne funktionalitet er beregnet som en hjælp, men kun en mindre del af C vil producere x86 assembler, der kan håndteres korrekt af primify. Det er nødvendigt manuelt at verificere korrektheden af den producerede x86prime kode.

prasm indkoder x86prime assembler programmer til et hexadecimalt format, som kan indlæses af den udleverede halvfærdige simulator.

prun simulerer x86prime programmer og generere en "sporingsfil". Denne fil kan indlæses af den udleverede halvfærdige simulator, hvor den kan bruges til at holde den ufærdige simulators opførsel op imod referencesimulatorens.

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<sup>1</sup>. Vi frigiver et "image" til en virtual maskine hvor værktøjet er pre-installeret og har de nødvendige biblioteker til oversættelse.

## 3 Kildetekst til simulator

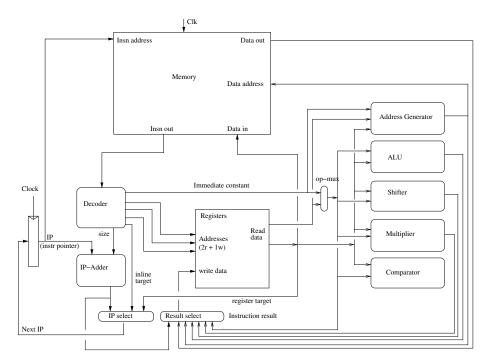
Den halvfærdige simulator som kan køre RET %r15, MOVQ \$imm, %rXX og MOVQ %rXX, %rYY instruktionerne, finder du i en zip-fil der distribueres sammen med denne opgavetekst.

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 implementerer præcis denne datavej, 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 Diverse regneenheder der passer til x86prime.
- memory.h, memory.c Lageret (Main memory).
- registers.h, registers.c Registerfilen.
- ip\_reg.h, ip\_reg.c IP-registeret (Program Counter).

<sup>&</sup>lt;sup>1</sup>Brug forum for kurset til at bede om hjælp til at løse problemer. Der er en sandsynlighed for at en medstuderende kender til det.



DIKU

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

#### Byggeklodser 3.1

Den halvfærdige simulator skal færdiggøres. I skal bruge sproget 'C' til at bygge en simulator som emulerer en faktisk elektronisk processor. Vi sætter nogle begrænsninger på hvordan I må bruge 'C' til denne implementation. Disse begrænsninger tvinger jer til at bruge 'C' som om det var en form for hardware-beskrivelses-sprog fremfor et generelt anvendeligt programmeringssprog. De begrænsinger vi vælger vil virke snærende, men de er essentielle for at sikre forståelse af hvordan maskinen virker og er opbygget.

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.), men kun på variable af typen bool. 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 arbejder på værdier af denne type.
- I må ikke bruge conditionals (if, else, ?-operatoren).

- I må ikke bruge løkker (for, while) udover hovedløkken der allerede er skrevet.
- I må ikke sætte en variabels værdi mere end én gang. Hver variabel repræsenterer en elektrisk leder (eller flere) og sådanne kan kun lede et stabilt signal i løbet af en given maskin-cyklus. 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 **Teoretisk opgave**

(En opgave om teoretisk simulering af en data cache kommer her senest søndag d. 29. september.)

#### 5 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.
- 40% Rapport som dokumenterer jeres implementation. Husk en rapport inkluderer:
  - Beskrivelse af hvordan jeres kode oversættes og hvordan jeres tests skal køre for at genskabe jeres resultater.
  - Diskusion af alle ikke-trivielle dele af jeres implementation og design beslutninger.
  - Beskrivelse alle tvetydige formuleringer, som I kan have fundet i opgave teksten.
  - Generel test af de implementerede dele af simulatores inklusiv tydeliggørelse af evt. mangler og afvigelser.
- 15% for løsning af den teoretiske opgave.

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:

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

 ${\tt x86prime}$  instructions and encoding

encoding.txt

Encoding:	Assembler	Operation
00000000 0000ssss 0001aaaa ddddssss 00100001 ddddssss 00110001 ddddssss 001111001 ddddssss	ret s op s,d movq s,d movq (s),d movq d,(s)	<pre>return from function call reg/reg arithmetic (see below) reg-&gt;reg copy load (memory -&gt; reg copy) store (reg -&gt; memory copy)</pre>
0100 cccc ddddssss pp32pp 0100 1110 dddd0000 pp32pp 0100 1111 00000000 pp32pp 0101 aaaa dddd0000 ii32ii 01100100 dddd0000 ii32ii 0111 1101 ddddssss ii32ii	<pre>cb<c> s,d,p call p,d jmp p op \$i,d movq \$i,d movq i(s),d movq d,i(s)</c></pre>	<pre>compare and continue at p if (see function call continue at p constant/reg arithmethic(see below) constant -&gt; register load (memory -&gt; reg copy) store (reg -&gt; memory copy)</pre>
10000001 ddddsss 10010010 dddd0000 zzzzvvvv 10010011 ddddsss zzzzvvvv 10100100 dddd0000 ii32ii 10100101 ddddsss ii32ii 10110110 dddd0000 zzzzvvvv ii32ii 10110111 ddddsss zzzzvvvv ii32ii	<pre>leaq (s),d leaq (,z,v),d leaq (s,z,v),d leaq i,d leaq i(s),d leaq i(,z,v),d leaq i(s,z,v),d</pre>	<pre>s -&gt; d z*v -&gt; d s+z*v -&gt; d i -&gt; d i+s -&gt; d i+z*v -&gt; d i+z*v -&gt; d i+s+z*v -&gt; d</pre>
Explanations:	p cb <c> φ1,α,p</c>	compare and continue at p if (see
aaaa indicates the kind of arithmetic operation. All operate on full 64 bits:  0000 add addition 0001 sub subtraction 0010 and bitwise and 0011 or bitwise or 0100 xor bitwise xor		
0101 mul unsigned multiplication 0110 sar shift arithmetic right (preserve topmost bit) 0111 sal shift arithmetic left (zero into lsb, do not preserve topmost bit) 1000 shr shift (logical) right (zero into topmost bit) 1001 imul signed multiplication		
d,s and z are registers:		
0000 %rax 1000 %r8 0001 %rbc 1001 %r9 0010 %rcx 1010 %r10 0011 %rdx 1011 %r11		

Computer Systems 2019 DIKU **Due:** Søndag, 6. Oktober, 10:00 A2: Simulering af x86prime - x86\_64 delmængde med udvidelser **Version 1** 

0100 %rbp 1100 %r12 0101 %rsi 1101 %r13 0110 %rdi 1110 %r14 0111 %rsp 1111 %r15

v is a scale factor encoded into the field vvvv in the form of a shift amount as follows:

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, but the
conditions in x86prime carry the same meaning as for x86.

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

Encoding Semantic
0000 e Equal
0001 ne Not equal

0010 <reserved>
0011 <reserved>

0100 l less (signed)

0101 le less or equal (signed)
0110 g greater (signed)

0111 ge greater or equal (signed)

1000 a above (unsigned)

1001 ae above or equal (unsigned)

1010 b below (unsigned)

1011 be below or equal (unsigned)

11xx <reserved>

Note that signed and unsigned comparisons are different.

call places the return address in a register instead of pushing it onto the stack. ret returns to the address in a register instead of popping it from the stack.