CompSys
Assignment A5: Simulering af x86prime

Anders Persson `tqc110` | Caroline Kierkegaard `qlj556` | Mikkel Willén `bmq419`

21. december 2021

and Casper Larsen.

Introduktion

I denne aflevering har vi færdiggjort en simulator for et mindre instruktionssæt defineret over x86prime. Vi fik udleveret en halvfærdig simulator, hvor vi udfyldte de givne TODO's og testede vores implementation.

Bruger-guide

For at køre vores program, skal man først have oversat sit .prime program til hex kode. Derefter kan man køre vores program, ved at bruge kommandoen

```
$ ./sim your_file.hex "adress in hexnotation"
```

For at køre testfilen, skal man først ind og ændre, hvor prun og prasm ligger på computeren i test.sh filen. Efter man har gjort dette, kan man køre testfilen, ved at skrive

```
$ bash test.sh
```

i src mappen.

Implementation

For at færdiggøre simulatoren var det nødvendigt at lave tre forbedringer.

1. Vi skulle bestemme længden af hver instruktion
2. Vi skulle få de instruktioner som tilgår lageret til at virke rigtigt
3. Og de instruktioner som ændrer afviklingen af programmet skulle implementeres

Længden på instruktionerne

Simulatoren som vi fik udleveret, antog at alle instruktioner var 2 bytes. Dette ville hurtigt skabe problemer, så snart der skulle udføres en instruktion af en anden længde.
Vi løste dette problem ved at lave en boolean for hver mulig længde af de forskellige instruktioner. Information om længden fandt vi i encoding.txt. Derefter kunne vi bruge disse bools til ved hjælp af hjælpefunktionerne or og use_if at tjekke længden og bruge den som størrelse.

```
bool len2 = is_return_or_stop ||
            is_reg_arithmetic ||
            is_reg_movq ||
            is_reg_movq_mem ||
            is_leaq2;
bool len3 = is_leaq3;
bool len6 = is_imm_arithmetic || is_imm_movq || is_imm_movq_mem || is_cflow || is_leaq6;
bool len7 = is_leaq7;
bool len10 = is_imm_cbranch;
val ins_size = or(use_if(len2, from_int(2)),
               or(use_if(len3, from_int(3)),
               or(use_if(len6, from_int(6)),
               or(use_if(len7, from_int(7)),
                 (use_if(len10, from_int(10)))))));
```

Skriv i lageret

Programmet kunne ikke skrive til lageret, med den kode vi fik udleveret. Vi tilføjede derfor to boolske værdier, til at holde styr på, om der skal skrives til lageret, eller om der skal læse fra lageret.

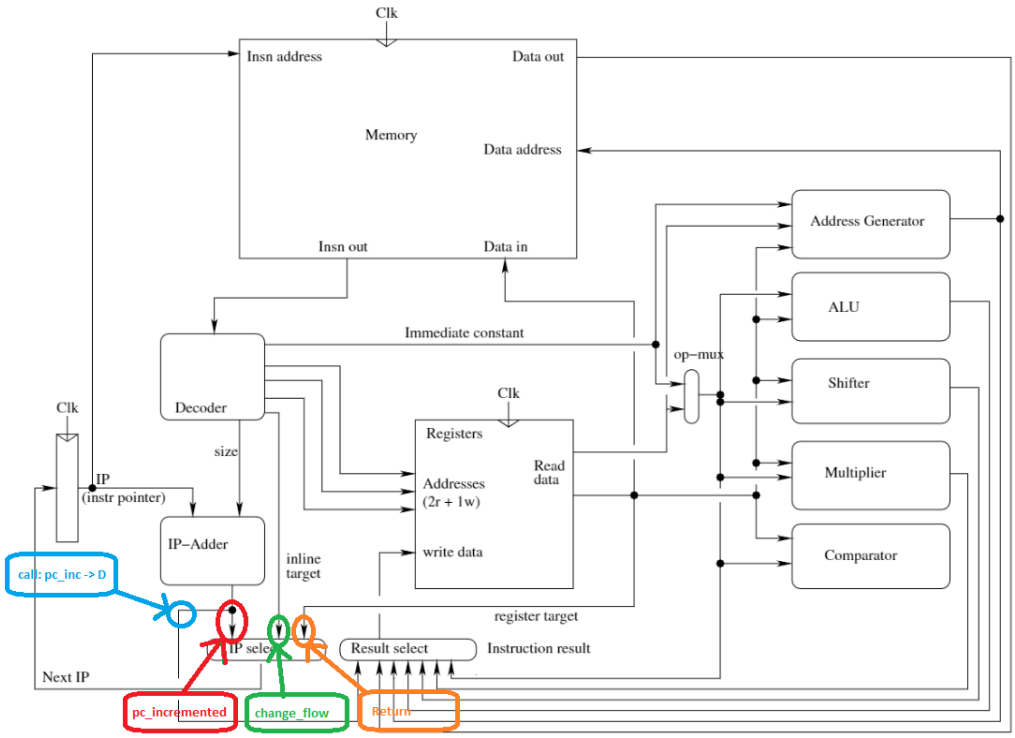
```
bool is_load = (is_reg_movq_mem || is_imm_movq_mem) && is_minor_load;
bool is_store = (is_reg_movq_mem || is_imm_movq_mem) && is_minor_store;
```

Disse værdier bliver brugt længere nede i koden i nogle funktioner, som allerede var blevet lavet.

Derudover skulle vi også sørge for, at der blev skrevet en værdi til lageret, når der bliver kaldt call. Efter call skal programmet nemlig gå tilbage til det sted, hvor den var nået til. Det gjorde vi ved at tilføje en linje, som tjekker om instruktionen er call, og hvis den er det, sætter vi datapath_result, til den værdier, som programmet er noget til i koden. Der er så en anden allerede implementeret funktion, som håndtere, at skrive denne værdi til lageret.

```
val datapath_result = or(use_if(use_compute_result, compute_result),
                      or(use_if(is_call, pc_incremented),
                         use_if(is_load, mem_out)));
```

Nedenfor kan ses et billede af, hvad de forskellige værdier, som IP-select skal vælge mellem, kommer fra.



Figur 1: Diagram som viser datavejen for en simpel enkelt-cyklus processor - nu med vores pile også

Her kan man også se, hvordan maskinen gemmer værdien for pc_incremented, i tilfælde af den kører en call instruktion.

Afvikling af programforløb

Den udleverede simulator kunne heller ikke håndtere instruktionerne JMP, RET, CALL og CBcc, men kun udføre en sekvens af fortløbende instruktioner. Vi tilføjede val-væddierne for de forskellige instruktioner, som skulle give adressen på den næste instruktion. Derefter brugte vi funktionerne or og use_if til at afgøre hvilken instruktion der blev kaldt og så vælge det rigtige sted at hoppe hen til.
JMP, CALL og CBcc hopper alle til en 32 bit-værdi som er gemt i P. RET returnere fra et funktionskald og springer til adressen ssss, som er gemt i c-koden som reg_out_b.

```
val pc_incremented = add(pc, ins_size);
val pc_jmp         = sext_imm_p;
val pc_call        = sext_imm_p;
val pc_conditional = sext_imm_p;
val pc_return      = reg_out_b;

val pc_next = or(use_if(is_normal, pc_incremented),
              or(use_if(is_jmp, pc_jmp),
              or(use_if(is_call, pc_call),
```

```
or(use_if(is_cond_true, pc_conditional),
  (use_if(is_return, pc_return))))));
```

Testing

Vi har testet vores implementation på et par forskellige `.prime` filer. Vi har lavet en del meget simple tests, så gerne bare skulle teste nogle få enkelte dele af vores implementation. Det er fx tests af subq og addq, for at teste om aritmetiske opperationer virker efter hensigten. Derudover har vi lavet et par tests, hvor programmet sammenligner to resultater, og springer til et bestemt sted i koden, alt efter, om den boolske logik er korrekt eller ej. Da vi var sikre på disse tests fungerede, og at de gav det rigtige resultat, lavede vi nogle større tests, som skulle teste mangler flere ting på en gang. Vi mente dette var en god ide, da vi gerne ville være sikre på, forskellige kombinationer af kode også fungerede efter hensigten.

Utilstrækkeligheder og potentielle problemer

Man kunne med fordel, for at være sikker på oversætteren virker korrekt, lave tests for alle kombinationer af major og minor opcodes. Dette ville give en rigtigt godt billede af, om stort set al funktionaliteten i vores oversætter virker korrekt. Man ville på denne måde, kunne teste, om programmet korrekt finder og ændre pegeren for næste instruktion, og derudover også om programmet skriver til memory, når den burde. Man ville også kunne bestemme, om programmet korrekt hopper til de rigtige steder i koden, ved de forskellige opperationer, som ændre pegeren for næste instruktion, ved at man har givet den en adresse på næste instruktion, på et tidspunkt i sin kode. Det ville dog være et meget stort arbejde, at lave tests for det hele, da der er rigtigt mange kombinationer af major og minor opcodes. Der er også mange af disse instruktioner, som kræver flere tests, da der er mulighed for flere forskellige udfald. Dette er specielt alle sammenligningsinstruktionerne, som enten kan gå videre til næste instruktion eller skal springe et bestemt sted hen i koden. Vi håber dog, at vi har testet det nogenlunde, med vores større funktioner, da disse går frem og tilbage i koden en del gange, og derfor tester mange af sammenligningerne med forskellige tal, og også både, skal springe et andet sted hen i koden, men også bare skal gå videre til næste instruktion.

Da alle vores tests oversætter programmet korrekt, og da vi mener vi har testet de mest essentielle kald af instruktioner, vil vi mene, at vores program fungerer korrekt, og at der derfor ikke er nogle utilstrækkeligheder for vores implementation af oversætter.

Konklusion

I denne aflevering fik vi udleveret en halvfærdig simulator som vi har færdiggjort ved hjælp af forskellige hjælpefunktioner i de udleverede c-filer, information fra encoding.txt og med de begrænsninger som blev stillet til opgaven. Vi har fået simulatoren til at håndtere forskellige længde instruktioner, skrive rigtigt til lageret og implementeret instruktioner JMP, RET, CALL og CBcc som ændrer programmets afvikling. Efter hver af disse ændringer har vi testet vores program, og da disse tests er succesfulde, mener vi at vi har løst opgaven og færdiggjort simulatoren.

Den teoretiske del

4.1

LFB0:
%rsp er en pointer (stack pointer)
%rax er en long

%rdx er en long
%rsi er en pointer
%r11 er en pointer
%rcx er både en pointer og en long
%rdi er en pointer

LFB1:
%rsp er en pointer (stack pointer)
%r11 er en pointer
%rdi er en pointer
%rcx er en long
%r8 er en pointer
%rax er en long
%rsi er en pointer
%rdx er en pointer

4.2

Når fx c skal kalde retur, udfører assembly disse 3 linjer:

```
movq (%rsp), %r11
addq $8, %rsp
ret %r11
```

Og i starten når vi kalder funktionen så udføres disse linjer.

```
subq $8, %rsp
movq %r11, (%rsp)
```

4.3

LFB0:
Den starter med at køre LBF0 og når den når til L2 tjekker den om %rdx er mindre end eller %rax. Hvis dette holder hopper den til L4, ellers bliver den ved med at køre L2.
I L4 returnerer den returværdien i %r11

LBF1:
Den starter med at køre LFB1, hvorefter den hopper til L6. I L6 tjekker den om %r8 $\leq$ %rdi og hvis dette holder hopper den til L10, som returnerner returværdien.
Hvis dette tjek ikke holder, fortsætter den i L6 og tjekker om %rax $\geq$ %rcx. Hvis dette er sandt hopper den til r7 og ellers forsætter den i L6. I begge scenarier forsætter koden med at køre L8 og derefter L6, indtil at det første tjek i L6 er sandt og den kører L10 som returnerer.

4.4

Alle cb kaldene og jmp bestemmer, hvor i koden vi skal læse fra. Se 4.3 for uddybning.

4.5

LFB0:

```
void copyArray(long* array, long* copy, long length) {
    copy = Malloc(sizeof(array));
    for (long i = 0; i < length - 1; i++) {
        copy[i] = array[i];
    }
}
```

LFB1:

```
void compareArray(long* array1, long* array2, long* compArray, long length) {
    compArray = Malloc(sizeof(array));
    for (long i = 0; i < length - 1; i++) {
        if (array1[i] <= array2[i]) {
            compArray[i] = array1[i];
        } else {
            compArray[i] = array2[i];
        }
    }
}
```