Fra C til RISCV - et minimalt setup

De næste fire uger skal vi tættere på RISCV. Vi starter med en (ny) kobling fra C til RISCV.

- Vi vil bruge en kryds-oversætter fra C til RISCV
- En simulator for RISCV
- Et ultra-minimalistisk bibliotek i stedet for standard biblioteket
    - Så vi har ikke noget printf() eller malloc()!
    - Men vi har noget andet....

Fra C til RISCV

Vi har installeret en krydsoversætter på en server på DIKU.

- I kataloget "resources/tiny_riscv" i kursets offentlige "repo" findes nogle scripts der skal bruges for at tilgå oversætteren.
- Samme katalog indeholder også nogle eksempler på C programmer der kan oversættes.

Det er også muligt at lave en lokal installation af en tilsvarende oversætter:

- Mac: https://github.com/riscv-software-src/homebrew-riscv
- Linux: https://github.com/stnolting/riscv-gcc-prebuilt
- Windows: Brug WSL og Linux installation

Men vi anbefaler brug af kursets oversætter.

En RISCV simulator

- Vi bruger vores egen
- Findes i "tools/riscv-sim/"
- Der er eksekverbare udgaver for Linux og Mac (både x86 or ARM)
- På Windows kan bruges WSL og Linux udgaven af simulatoren

Et minimalt bibliotek

- Til brug i resten af kurset har vi et meget lille bibliotek.
- Der er ikke noget standard C bibliotek. Tough Luck!
- Biblioteket udgøres af filerne lib.h og lib.c som findes i kursets offentlige repo i kataloget resources/tiny_riscv.
- For at oversætte et C-program skal man medtage "lib.c" i oversættelsen.

SKAL VI PRØVE?

Et minimalt bibliotek

API:

```
#ifndef __LIB_H__
#define __LIB_H__

#define NULL 0

char inp();
void outp(char);
void terminate(int status);
void print_string(const char* p);
void read_string(char* buffer, int max_chars);
unsigned int str_to_uns(const char* str);
int uns_to_str(char* buffer, unsigned int val);
void* allocate(int size);
void release(void* mem);
#endif
```

Der er ikke mange kommentarer - men det bliver ikke simplere!

GÆTTEKONKURRENCE: Hvad gør disse funktioner?

Eksempel - C program der bruger biblioteket

```
#include "lib.h"

unsigned int fib(unsigned int arg) {
  if (arg < 2) return arg;
  return fib(arg - 1) + fib(arg - 2);
}

int main(int argc, char* argv[]) {
  if (argc < 2) {
    print_string("fib() missing arguments\n");
    terminate(1);
  }
  char buffer[20];
  unsigned int arg = str_to_uns(argv[1]);
  print_string("fib(");
  print_string(argv[1]);
  print_string(") = ");
  unsigned int res = fib(arg);
  uns_to_str(buffer, res);
  print_string(buffer);
  print_string("\n");
  return 0;
}
```

Eksempel - fortsat

Observationer:

- Man kan overføre argumenter via arcg/argv som sædvanligt
- Vi kan se at programmet konverterer mellem heltal og strenge
  - str_to_uns()
  - uns_to_str()
- Der kunne også se ud til at være mulighed for I/O
  - print_string()

Oversættelse

Simpelt - husk bare at tage "lib.c" med:

```
./gcc.py -O2 hello.c lib.c -o hello.elf
```

Oversættelsen genererer en ELF RISC-V eksekverbar. Den kan køres af vores simulator.

Hvis du installerer din egen kryds-oversætter lokalt skal du vide at det er lidt tricky at få de rette flag til krydsoversætteren

- Den skal bruge vores bibliotek i stedet for standard biblioteket
- Den skal kun generere kode som simulatoren kan klare
- Det klarer vores script.

Men hvis du installerer din egen er her alle de magiske options, du skal give:

```
./gcc -march=rv32im -mabi=ilp32 -fno-tree-loop-distribute-patterns
      -O1 fib.c lib.c -static -nostartfiles -nostdlib -o fib.elf
```

Sammen med programeksemplerne findes en Makefile der vil oversætte og linke med vores særlige lille bibliotek

Disassemblering

Man kan "dis-assemble" ELF filen

```
./objdump.py -S fib.elf
```

Det kan se således ud:

```
000100c4 <main>:
   100c4:       fd010113                addi    sp,sp,-48
   100c8:       02112623                sw      ra,44(sp)
   100cc:       02812423                sw      s0,40(sp)
   100d0:       02912223                sw      s1,36(sp)
...
```

RISCV assembler for fib.c

Sådan her ser vores fib program ud i RISCV assembler:

```
00010094 <fib>:
   10094:       ff010113                addi    sp,sp,-16
   10098:       00112623                sw      ra,12(sp)
   1009c:       00812423                sw      s0,8(sp)
   100a0:       00912223                sw      s1,4(sp)
   100a4:       00050413                mv      s0,a0
   100a8:       00100793                li      a5,1
   100ac:       00a7fe63                bgeu    a5,a0,100c8 <fib+0x34>
   100b0:       fff50513                addi    a0,a0,-1
   100b4:       fe1ff0ef                jal     ra,10094 <fib>
   100b8:       00050493                mv      s1,a0
   100bc:       ffe40513                addi    a0,s0,-2
   100c0:       fd5ff0ef                jal     ra,10094 <fib>
   100c4:       00a48533                add     a0,s1,a0
   100c8:       00c12083                lw      ra,12(sp)
   100cc:       00812403                lw      s0,8(sp)
   100d0:       00412483                lw      s1,4(sp)
   100d4:       01010113                addi    sp,sp,16
   100d8:       00008067                ret
```

Hvordan var det nu det var ... det der RISCV ?

RISCV Instruktioner

RISCV er en RISC arkitektur (RISC = Reduced Instruction Set Computer). Surprise!

Der er følgende grupper af instruktioner

- Aritmetik - arbejder udelukkende med registre
- Lagertilgang - eneste instruktioner der kan tilgå lageret
- Kontrol af programforløb - arbejder udelukkende med registre
- Systemkald
- Diverse

I praksis defineres RISC ved at der er særlige instruktioner der kun har til formål at tilgå lageret og alle andre instruktioner arbejder kun med registre. Derfor kaldes en RISC undertiden også for en "Load-Store arkitektur"

RISCV Assembler - Aritmetik

Der findes to slags aritmetiske instruktioner

- register/konstant operationer: rd <- op(rs1, imm)

`addi, slli, slti, sltiu, xori, srli, srai, ori, andi.`

- register/register operationer: rd <- op(rs1, rs2)

`add, sub, sll, slt, sltu, xor, srl, sra, or, and.`

Og to specielle til at forme konstanter på større end 12 bit:

`lui, auipc`

Detaljer: https://github.com/riscv/riscv-isa-manual/releases/download/20240411/unpriv-isa-asciidoc.pdf

RISCV Assembler - Lagertilgang

Instruktioner til at læse fra lageret:

`lb, lh, lw, lbu, lhu`

Og til at skrive

`sb, sh, sw`

Alle instruktioner der tilgår lageret beregner den adresse de tilgår, som en sum af et register og en 12-bit fortegnsbefængt konstant.

Hvorfor er der flere instruktioner til at læse end til at skrive?

Detaljer: https://github.com/riscv/riscv-isa-manual/releases/download/20240411/unpriv-isa-asciidoc.pdf

RISCV Assembler - Kontrol

Vi har 6 betingede hop:

`beq, bne, blt, bge, bltu, bgeu`

Og 2 til at lave både kald og retur:

`jal, jalr`

Detaljer: https://github.com/riscv/riscv-isa-manual/releases/download/20240411/unpriv-isa-asciidoc.pdf

RISCV Assembler - Systemkald

Systemkald består i at placere diverse argumenter i registre og så udføre instruktionen 'ecall'. På en rigtig maskine ville dette føre til et kald af kode i styresystemet. Denne kode ville så implementere de forskellige systemkald.

I vores lille verden er der ikke noget styresystem og simulatoren implementerer i stedet de forskellige systemkald direkte. For eksempel

```
outp:
    li a7,2   # systemkald nummer 2 udskriver indholdet af a0 som et tegn
    ecall
    ret
```

Detaljer: https://github.com/riscv/riscv-isa-manual/releases/download/20240411/unpriv-isa-asciidoc.pdf

RISCV Assembler - Pseudo instruktioner

Pseudoinstruktioner er specielle omskrivninger af andre instruktioner, som kan gøre programmer nemmere at læse eller skrive:

```
beqz, bnez  Sammenligning med 0 og betinget hop
seqz, snez  Sammenligning med 0
li, la      Placer konstant i register
mv          Move fra et register til et andet
j,jr        Ubetinget hop (evt til register)
call        Funktionskald
ret         Retur fra funktionskald
```

Pseudoinstruktioner er implementeret ved hjælp af en eller flere virkelige instruktioner.

Detaljer: https://github.com/riscv/riscv-isa-manual/releases/download/20240411/unpriv-isa-asciidoc.pdf

En gang til:

```
unsigned int fib(unsigned int arg) {
  if (arg < 2) return arg;
  return fib(arg - 1) + fib(arg - 2);
}

00010094 <fib>:
   10094:  ff010113    addi    sp,sp,-16          # ...prolog...
   10098:  00112623    sw      ra,12(sp)
   1009c:  00812423    sw      s0,8(sp)
   100a0:  00912223    sw      s1,4(sp)
   100a4:  00050413    mv      s0,a0              # if (arg < 2)...
   100a8:  00100793    li      a5,1
   100ac:  00a7fe63    bgeu    a5,a0,100c8 <.L1>
   100b0:  fff50513    addi    a0,a0,-1           #   s1 = fib(arg - 1)
   100b4:  fe1ff0ef    jal     ra,10094 <fib>
   100b8:  00050493    mv      s1,a0
   100bc:  ffe40513    addi    a0,s0,-2           #   a0 = fib(arg - 2)
   100c0:  fd5ff0ef    jal     ra,10094 <fib>
   100c4:  00a48533    add     a0,s1,a0           #   a0 = a0 + s1
.L1:                                           # }
   100c8:  00c12083    lw      ra,12(sp)          # ...epilog...
   100cc:  00812403    lw      s0,8(sp)
   100d0:  00412483    lw      s1,4(sp)
   100d4:  01010113    addi    sp,sp,16
   100d8:  00008067    ret
```

Biblioteket - opstart

Når simulatoren har indlæst koden i .riscv filen vil den starte simulationen efter "_start" symbolet. Øverst i lib.c findes den kode, der så udføres. Koden er skrevet i en særlig "inline assembler" format, som gør at den kan placeres i en 'C' fil:

```
asm("  .globl _start");
asm("_start:");
asm("  li a0, 0x1000000");  // set start of stack (which grows in opposite direction)
asm("  mv sp, a0");
asm("  li a0, 0x2000000");  // set start of heap area
asm("  call init_heap");
asm("  li a0, 0x1000000");  // arg area is right after stack (filled by simulator)
asm("  call args_to_main");
asm("  call terminate");
```

Koden placerer stakken fra 0x1000000 og ned, heapen fra 0x2000000 og op, og et område til kommandolinieparametere fra 0x1000000 og op.

I args_to_main() initialiseres argc/argv fra området med kommandolinieparametre og så kaldes main().

Biblioteket - I/O

Biblioteket indeholder 4 simple rutiner til input/output:

- inp() indlæser et enkelt tegn
- outp() udskriver et enkelt tegn
- print_string() udskriver en nul-termineret streng
- read_string() indlæser en streng

inp() og outp() bruger inline assembler

Biblioteket - Andre funktioner

Biblioteket indeholder 2 simple rutiner til konvertering mellem heltal og strenge.

- str_to_uns() konverterer streng til unsigned.
- uns_to_str() konverterer unsigned til streng.

Endeligt har vi 2 funtioner til administration af lager

- allocate() vil allokere en blok fra heapen
- release() vil frigive en tidligere allokeret blok

De to funktioner er ret forsimplede og kan kun understøtte allokeringer mindre end 4K.

Simulering

Resultatet af oversættelsen (en riscv .elf fil) kan simuleres

```
./sim fib.elf -- 7
fib(7) = 13

Simulated 930 instructions in 23 ticks (40.434783 MIPS)
```

Simulatoren vil kopiere kommandolinie-argumenter efter "--", i det her tilfælde "7", ind i det simulerede lager. Derfra vil koden vi tidligere præsenterede hente det og sætte argc/argv op, så det simulerede program kan læse argumenterne i main().

Simulering - Sporingsudskrift

Det er muligt at bede simulatoren om en sporingsudskrift:

```
./sim fib.elf -l log -- 7
```

Vil producere filen 'log'. Den indeholder først et echo af indlæsningen fra fib.elf, derpå en linie for hver eneste udført instruktion. Her er et udsnit af fib() der kalder sig selv rekursivt:

```
157 =>  10094 : ff010113     fib:   ADDI sp, sp, -16        R[ 2] <- ffffb0
158     10098 : 00112623            SW ra, 12(sp)                  1012c    ->  Mem[fff
159     1009c : 00812423            SW s0, 8(sp)                   1000004  ->  Mem[fff
160     100a0 : 00912223            SW s1, 4(sp)                   7        ->  Mem[fff
161     100a4 : 00050413            ADDI s0, a0, 0         R[ 8] <- 7
162     100a8 : 00100793            ADDI a5, zero, 1       R[15] <- 1
163     100ac : 00a7fe63            BGEU a5, a0, 100c8 {_}
164     100b0 : fff50513            ADDI a0, a0, -1        R[10] <- 6
165     100b4 : fe1ff0ef            CALL 10094             R[ 1] <- 100b8
166 =>  10094 : ff010113     fib:   ADDI sp, sp, -16        R[ 2] <- ffffa0
167     10098 : 00112623            SW ra, 12(sp)                  100b8    ->  Mem[fff
168     1009c : 00812423            SW s0, 8(sp)                   7        ->  Mem[fff
169     100a0 : 00912223            SW s1, 4(sp)                   7        ->  Mem[fff
170     100a4 : 00050413            ADDI s0, a0, 0         R[ 8] <- 6
171     100a8 : 00100793            ADDI a5, zero, 1       R[15] <- 1
172     100ac : 00a7fe63            BGEU a5, a0, 100c8  {_}
173     100b0 : fff50513            ADDI a0, a0, -1        R[10] <- 5
174     100b4 : fe1ff0ef            CALL 10094             R[ 1] <- 100b8
```

Sporingsudskrift, forklaring:

```
Instruktionsnummer siden start
|   Markering af indhop
|   |   Addresse
|   |   |       Indkodning
|   |   |       |            Disassembly
|   |   |       |                            Hop ikke taget/taget
|   |   |       |                            |   Effekt
|   |   |       |                            |   |
157 =>  10094 : ff010113     fib:   ADDI sp, sp, -16        R[ 2] <- ffffb0
158     10098 : 00112623            SW ra, 12(sp)                  1012c    ->  Mem[fff
159     1009c : 00812423            SW s0, 8(sp)                   1000004  ->  Mem[fff
160     100a0 : 00912223            SW s1, 4(sp)                   7        ->  Mem[fff
161     100a4 : 00050413            ADDI s0, a0, 0         R[ 8] <- 7
162     100a8 : 00100793            ADDI a5, zero, 1       R[15] <- 1
163     100ac : 00a7fe63            BGEU a5, a0, 100c8 {_}
164     100b0 : fff50513            ADDI a0, a0, -1        R[10] <- 6
165     100b4 : fe1ff0ef            CALL 10094             R[ 1] <- 100b8
```