

# 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 ikke noget printf eller malloc!

# Fra C til RISCv

## installation

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

# En RISC-V simulator

- Vi bruger vores egen
- Findes i kursets public repo
- Under tools/simulator
- Vælg den eksekverbare der matcher din platform

# 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`.

# 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 simple!

# 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 `argc/argv` som sædvanligt
- Vi kan se programmet konvertere 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

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

Sådan ser det ud:

```
./gcc -march=rv32im -mabi=ilp32 -fno-tree-loop-distribute-patterns  
      -mno-relax -O1 fib.c lib.c -static -nostartfiles -nostdlib -o fib.riscv  
./objdump -s -w fib.riscv > fib.dis  
./objdump -S fib.riscv >> fib.dis
```

Sammen med programeksemplerne findes en Makefile der vil oversætte, linke med biblioteket og konvertere programmet til et format vores simulator kan læse.

De filer vores simulator kan læse kaldes for .dis filer.



# RISCV output

En .dis fil indeholder to ting:

- Et dump af lagerets indhold ved opstart

```
10104 97000000 e780c010 37050100 1305c54e .....7.....N
10114 97000000 e780c00f 13850400 97000000 .....
10124 e78040f5 93050500 1305c100 97000000 ..@.....
...
```

- Disassembly af kode i lageret

```
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:

```
00010074 <fib>:
10074:    ff010113      addi    sp,sp,-16
10078:    00112623      sw      ra,12(sp)
1007c:    00812423      sw      s0,8(sp)
10080:    00912223      sw      s1,4(sp)
10084:    00050413      mv      s0,a0
10088:    00100793      li      a5,1
1008c:    02a7f263      bgeu    a5,a0,100b0 <fib+0x3c>
10090:    fff50513      addi    a0,a0,-1
10094:    00000097      auipc   ra,0x0
10098:    fe0080e7      jalr    -32(ra) # 10074 <fib>
1009c:    00050493      mv      s1,a0
100a0:    ffe40513      addi    a0,s0,-2
100a4:    00000097      auipc   ra,0x0
100a8:    fd0080e7      jalr    -48(ra) # 10074 <fib>
100ac:    00a48533      add     a0,s1,a0
100b0:    00c12083      lw      ra,12(sp)
100b4:    00812403      lw      s0,8(sp)
100b8:    00412483      lw      s1,4(sp)
100bc:    01010113      addi    sp,sp,16
100c0:    00008067      ret
```

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

# RISCV Assembler - Aritmetik

Der findes to slags aritmetiske instruktioner

- register/konstant operationer: `id <- 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://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.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://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.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://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.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 at 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
```

Detaljer: <https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.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)
ret	Retur fra funktionskald

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

Detaljer: <https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf>

# Biblioteket - opstart

Når simulatoren har indlæst koden i .dis 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 kommandolinieparametre 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_unsigned()` konverterer streng til unsigned.
- `unsigned_to_str()` konverterer unsigned til streng.

Endeligt har vi 2 funktioner 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

```
./sim fib.dis -- 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 præsentered hente det og sætte argc/argv op, så det simulerede program kan læse argumenterne i main().

# Simulering - Springudskrift

Det er muligt at bede simulatoren om en springudskrift:

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

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

```
172 => 10074 : ff010113      addi      sp,sp,-16      R[ 2] <- fffffb0
173     10078 : 00112623      sw        ra,12(sp)      10128      -> Mem[ffffbc]
174     1007c : 00812423      sw        s0,8(sp)      1000004      -> Mem[ffffb8]
175     10080 : 00912223      sw        s1,4(sp)      7          -> Mem[ffffb4]
176     10084 : 00050413      mv        s0,a0          R[ 8] <- 7
177     10088 : 00100793      li        a5,1          R[15] <- 1
178     1008c : 02a7f263      bgeu      a5,a0,100b0    {_-}
179     10090 : fff50513      addi      a0,a0,-1      R[10] <- 6
180     10094 : 00000097      auipc     ra,0x0        R[ 1] <- 10094
181     10098 : fe0080e7      jalr      -32(ra)       R[ 1] <- 1009c
182 => 10074 : ff010113      addi      sp,sp,-16      R[ 2] <- ffffa0
183     10078 : 00112623      sw        ra,12(sp)      1009c      -> Mem[ffffac]
184     1007c : 00812423      sw        s0,8(sp)      7          -> Mem[ffffa8]
185     10080 : 00912223      sw        s1,4(sp)      7          -> Mem[ffffa4]
186     10084 : 00050413      mv        s0,a0          R[ 8] <- 6
187     10088 : 00100793      li        a5,1          R[15] <- 1
```

# Sporingsudskrift, forklaring:

Instruktionsnummer siden start	Markering af indhop	Adresse	Indkodning	Disassembly	Hop ikke taget/taget	Effekt	
177		10088 :	00100793	li a5,1		R[15] <- 1	
178		1008c :	02a7f263	bgeu a5,a0,100b0	{_}		
179		10090 :	fff50513	addi a0,a0,-1		R[10] <- 6	
180		10094 :	00000097	auipc ra,0x0		R[ 1] <- 10094	
181		10098 :	fe0080e7	jalr -32(ra)		R[ 1] <- 1009c	
182 =>		10074 :	ff010113	addi sp,sp,-16		R[ 2] <- ffffa0	
183		10078 :	00112623	sw ra,12(sp)		1009c	-> Mem[ffffac]