

RISCV - Mellem oversætter og hardware

Maskinsprog er et kompromis mellem hardware og software. Et godt maskinsprog gør det nemt at implementere såvel hardware som oversætter. Tidligere var et maskinsprog godt, hvis det var (relativt)) nemt for en programmør at udtrykke sig i. Nu er det vigtige at oversætteren kan bruge det godt.

Så lad os gå bare en lille smule i detaljer med hvordan oversætteren oversætter fra C til symbolsk maskinkode.

Oversættelse af simple udtryk

Oversætteren vil som udgangspunkt analysere hele funktioner og placere alle variable i registre. Det kaldes register-allokering. Der kan være så mange variable at der ikke er registre nok. Hvis det sker vil compileren generere ekstra kode til at flytte variable frem og tilbage mellem registre og celler i lageret. Det kaldes spill/fill.

Betragt følgende stump C kode

```
a = b + c;
```

Efter register allokering kan compileren udtrykke denne kode ved brug af simple operationer, der ligner en maskines instruktioner. De behøver dog ikke at matche maskinens instruktioner præcist i første omgang. Vi kunne få:

```
a = ADD(b, c)
```

hvor ADD jo sandsynligvis vil være en enkelt instruktion på enhver rimelig maskine. På RISC-V kunne det endelige resultat være:

```
ADD t2,t1,a5
```

hvor t2,t1 og a5 er registre som oversætteren har valgt til de forskellige variable.

Oversættelse af simple udtryk - II

Betragt følgende stump C kode igen

```
a = b + c;
```

Nogle maskiner tillader kun at man angiver 2 registre i en instruktion. Så må ovenstående implementeres ved 2 instruktioner:

```
MOVE t2,t1  
ADD t2,a5
```

Nyere maskiner, herunder RISC-V, har typisk instruktioner der kan angive 3 registre.

Oversættelse af simple udtryk - konstanter

Ofte indgår konstanter i udtryk:

```
a = b + 0x42424242
```

Oversætteren kunne internt have

```
a = ADD(b, 0x42424242)
```

Små konstanter er meget hyppige, så det vil være en god ide, hvis man kan angive konstanter direkte i instruktionen i stedet for et register. Men konstanter kan også være store. Hvis instruktionerne har en fast størrelse på 32 bit (f.eks.), så kan man åbenlyst ikke specificere en konstant på 32 bit i en enkelt instruktion.

For CISC maskiner har man ofte instruktioner der er ekstra lange til at håndtere større konstanter. På RISC maskiner (herunder RISC-V) bruger man i stedet flere instruktioner.

Store konstanter

Hvis en stor konstant indgår må oversætteren generere instruktioner som vil syntetisere den rette konstant.

```
a = b + 0x42424242
```

Vil for RISC-V blive til

```
LUI    t0,0x42424    # Load Upper Immediate (øverste 20 bits)
ADDI   t0,0x242      # Add nedre 12 bits, t0 indeholder nu 0x42424242
ADD    t2,a5,t0
```

Forskellige RISC maskiner understøtter forskellige størrelser af konstanter. På en MIPS (Forgænger til RISC-V) kunne LUI sætte de øverste 16 bits, og ADDI leverede de nederste 16 bits.

Adgang til lageret

Betragt

```
struct S { int a; int b; };  
struct S* p;  
p->a = p->b + 42;
```

Der findes maskiner som kan håndtere det i en enkelt instruktion, hvis 'p' er i et register. Men den fleksibilitet har sin pris (det vender vi tilbage til i en senere forelæsning). Fremgangsmåden for RISC maskiner er at håndtere lager tilgang ved dedikerede instruktioner:

```
LW    t0,4(a3)  
ADDI  t0,t0,42  
SW    t0,0(a3)
```

Notationen "4(a3)" kaldes en adresseringsmåde. Meningen er at at konstanten, her 4, og indholdet af registeret, her a3, lægges sammen for at give en adresse i lageret. Register a3 indeholder altså her pointeren 'p'.

Adgang til elementer i et array

Betragt

```
struct S { int a; int b; };  
struct S my_array[142];  
my_array[i] = my_array[j];
```

Lad os antage at en peger til my_array er i register a0, index i er i register s0 og index j er i register s1. Compileren vil så først forsimple ovenstående tildeling til følgende:

```
my_array_i = my_array + i * sizeof(struct S) # NOT 'C' - in 'C' scaling is implied  
my_array_j = my_array + j * sizeof(struct S) # NOT 'C' - in 'C' scaling is implied  
my_array_i->a = my_array_j->a;  
my_array_i->b = my_array_j->b;
```

Og med en størrelse på struct S på 8, vil multiplikationen blive implementeret med et venstre-skift.

```
SLLI    t0,s0,3  
SLLI    t1,s1,3  
LW      t2,0(t1)  
LW      t3,4(t1)  
SW      t2,0(t0)  
SW      t3,4(t0)
```

Indkodning af instruktioner til aritmetik

Valg i RISC-V: En instruktion fylder 32 bit. Små konstanter fylder 12 bit. Der er 32 registre, så de angives med 5 bit hver. Instruktioner angiver enten 3 registre eller 2 registre plus en lille konstant:

RISC-V har i alt 6 formater, og instruktionerne til aritmetik bruger formaterne R, I og U.

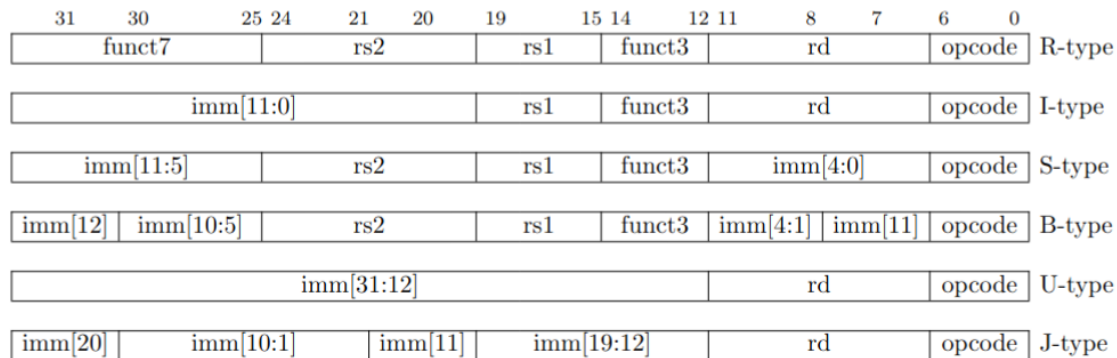


Figure 2.3: RISC-V base instruction formats showing immediate variants.

Aritmetik med en lille konstant

Konstanten må fylde op til 12 bits og er med fortegn!

imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI

Bemærk dog at skifte-instruktionerne tager mindre konstanter og bruger bit 30 til at skelne mellem SRLI og SRAI

Aritmetik med en stor konstant

Her bruges blot en enkelt instruktion, LUI, i kombination med instruktioner der tager små konstanter.



Lui sætter de øverste 24 bits som angivet i instruktioner, de nederste 12 bits sættes til 0.

Aritmetik med to registre

0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

Kommentarer?

Instruktioner til lagertilgang

Lageret tilgås udelukkende via "load" eller "store" instruktioner. Der er kun EN adresseringsmåde: adressen beregnes som register plus konstant. Der er 12 bit afsat til konstanten.

imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW

Bemærk den forskellige indkodning af vores 12-bit konstant.

Hvorfor mon det?

Programforløb - if-then-else

Forgreninger indkodes som betingede hop.

```
if (betingelse) { udtryk-hvis-sandt} else { udtryk-hvis-falsk}
```

Bliver til flg pseudokode

```
kode for betingelse
hvis ikke opfyldt hop til L1
kode for udtryk-hvis-sandt
hop til L2
L1:
kode for udtryk-hvis-falsk
L2:
```

Hvis der ikke er nogen else-gren nøjes man med

```
kode for betingelse
hvis ikke opfyldt hop til L1
kode for udtryk-hvis-sandt
L1:
```

Programforløb - if-then-else II

I RISC-V udtrykkes betingede hop i en enkelt instruktion, der sammenligner indholdet af to registre. Hvis betingelsen er mere kompliceret må der genereres instruktioner som evaluerer betingelsen indtil man har et resultat i et register, som man så kan sammenligne med et andet - evt x0. Eksempel

```
if (a < b) { a = b; } else { b = a; }
```

Kunne give

```
    blt b,a,L1
    addi a,b,0
    j L2
L1:
    addi b,a,0
L2:
```

Hvorfor "blt b,a,L1" og ikke "blt a,b,L1" ?

Programforløb - do-while

Også her bruges betingede hop:

```
do { krop } while (betingelse);
```

Bliver til:

```
l1:  
  kode for krop  
  kode for evaluering af betingelse  
  hvis (betingelse er sand) hop til l1
```

Her behøves ikke nogen yderligere instruktioner.

Programforløb - while

Her omskrives ofte til en betinget do-while, dvs:

```
while (betingelse) { krop }
```

Bliver til

```
    kode for betingelse
    hvis (betingelse IKKE er sand) hop til L1
L2:
    kode for krop
    kode for betingelse
    hvis (betingelse ER sand) hop til L2
L1:
```

Dog ses også:

```
    hop til L1
L2:
    kode for krop
L1:
    kode for betingelse
    hvis (betingelse er sand) hop til L2
```


Betingede hop - indkodning

De betingede hop indkodes alle i B-formatet

imm[12 10:5]	rs2	rs1	000	imm[4:1 11]	1100011	BEQ
imm[12 10:5]	rs2	rs1	001	imm[4:1 11]	1100011	BNE
imm[12 10:5]	rs2	rs1	100	imm[4:1 11]	1100011	BLT
imm[12 10:5]	rs2	rs1	101	imm[4:1 11]	1100011	BGE
imm[12 10:5]	rs2	rs1	110	imm[4:1 11]	1100011	BLTU
imm[12 10:5]	rs2	rs1	111	imm[4:1 11]	1100011	BGEU

Hvor hoppes hen? Den indkodede 12-bit konstant ganges med to, fortegnssforlænges og lægges til adressen på hop instruktionen. Bemærk at selvom B-formatet ligner I-formatet så er konstanten indkodet anderledes.

Programforløb - funktionskald

Ved funktionskald følges en kaldkonvention. Kaldkonventionen beskriver hvordan registre bruges ved funktionskald, herunder hvilke registre der bruges til argumenter og resultater og hvilke der garanteres bevaret henover kaldet.

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–7	t0–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Table 18.2: RISC-V calling convention register usage.

Lad os se på koden for fib() en gang til:

```
fib:
    addi    sp,sp,-16      # prolog - gem registre ihht kaldkonvention
    sw      ra,12(sp)
    sw      s0,8(sp)
    mv      s0,a0          # if (arg < 2)...
    li      a5,1
    bleu    a0,a5,.L1
    sw      s1,4(sp)       # preserve callers s1, we're gonna use s1 ourselves
    addi    a0,a0,-1       # a0 = fib(arg-1)
    call    fib
    mv      s1,a0          # s1 = a0, use s1 here to save a0
    addi    a0,s0,-2       # a0 = fib(arg-2)
    call    fib
    add     a0,s1,a0        # a0 = a0 + s1
    lw      s1,4(sp)       # restore s1, we're done using it
.L1:
    lw      ra,12(sp)      # epilog - retabler gemte registre
    lw      s0,8(sp)
    addi    sp,sp,16
    jr      ra             # retur fra funktion
```

Prolog og Epilog administrerer stakken, allokerer plads til registre der skal gemmes, og gemmer registre i henhold til kaldkonventionen.

Kald - indkodning

imm	20	10:1	11	19:12	rd	1101111	JAL
-----	----	------	----	-------	----	---------	-----

Funktionskald indkodes ofte med JAL instruktionen. Den hopper til en adresse der beregnes ved summen af JAL instruktionens egen adresse og en 20 bit konstant. Ved funktionskald bruges x1 ("ra") som destinationsregister. Samme instruktion bruges også til ubetingede hop, her angives x0 ("zero") som destinationsregister.

Hvis den kaldte funktion (eller hop destination) er for langt væk bruges i stedet to instruktioner som opbygger adressen i et register:

imm[31:12]		rd		0010111		AUIPC		
imm[11:0]		rs1		000		rd	1100111	JALR

```
AUIPC x5,upper_20_bit_displacement  
JALR x1,x5,lower_12_bit_displacement
```

AUIPC er som LUI, bortset fra at den indlejrede konstant lægges til adressen på instruktionen selv, før den skrives til de øverste 20 bit af destinationsregisteret.

retur - indkodning

Retur er såmænd genbrug af JALR, denne gang med en indlejret konstant der er 0, destinationsregister x0 ("zero") og kilderegister x1 ("ra").

Det er kombinationen af indlejret konstant, destinations-register og kilde-register der angiver hvad JALR er:

<code>rd == x1</code>	Funktionskald
<code>rd == x0, rs1 == x1, imm == 0</code>	Retur
<code>rd == x0, rs1 != x1</code>	Indirekte/Beregnet hop

Godbolt

Jeg anbefaler at I prøver <https://godbolt.org/>

Her kan man afprøve hvordan små funktioner oversættes af et væld af forskellige compilere til et væld af arkitekturer. Mere nørdet bliver det næsten ikke....

Tilbage til vores minimale Bibliotek - 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 kommandolinieparametre fra 0x1000000 og op.

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

Simulering

NB! Dette virker ikke før vi har fixet et problem med simulatoren.

```
./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 i `args_to_main()` hente argumenter og sætte `argc/argv` op, så det simulerede program kan læse argumenterne i `main()`.

Simulering - Sporningsudskrift

Det er muligt at bede simulatoren om en sporningsudskrift:

```
./sim fib.riscv -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]