31 30 29 28 27 26 25	24 23 22 21 20	19 18 17 16 15	14 13 12	11 10 9 8 7	6 5 4 3 2 1 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	rd	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11|19:12]				rd	opcode	J-type

Zbb: “Basic bit-manipulation” Extension

31 25	24 20	19 15	14 12	11 7	6 0	
0 1 0 0 0 0 0	rs2	rs1	1 1 1	rd	0 1 1 0 0 1 1	ANDN
0 1 0 0 0 0 0	rs2	rs1	1 1 0	rd	0 1 1 0 0 1 1	ORN
0 1 0 0 0 0 0	rs2	rs1	1 0 0	rd	0 1 1 0 0 1 1	XNOR
0 1 1 0 0 0 0	0 0 0 0 0	rs1	0 0 1	rd	0 0 1 0 0 1 1	CLZ
0 1 1 0 0 0 0	0 0 0 0 1	rs1	0 0 1	rd	0 0 1 0 0 1 1	CTZ
0 1 1 0 0 0 0	0 0 0 1 0	rs1	0 0 1	rd	0 0 1 0 0 1 1	CPOP
0 0 0 0 1 0 1	rs2	rs1	1 1 0	rd	0 1 1 0 0 1 1	MAX
0 0 0 0 1 0 1	rs2	rs1	1 1 1	rd	0 1 1 0 0 1 1	MAXU
0 0 0 0 1 0 1	rs2	rs1	1 0 0	rd	0 1 1 0 0 1 1	MIN
0 0 0 0 1 0 1	rs2	rs1	1 0 1	rd	0 1 1 0 0 1 1	MINU
0 1 1 0 0 0 0	0 0 1 0 0	rs1	0 0 1	rd	0 0 1 0 0 1 1	SEXT.B
0 1 1 0 0 0 0	0 0 1 0 1	rs1	0 0 1	rd	0 0 1 0 0 1 1	SEXT.H
0 0 0 0 1 0 0	0 0 0 0 0	rs1	1 0 0	rd	0 1 1 0 0 1 1	ZEXT.H
0 1 1 0 0 0 0	rs2	rs1	0 0 1	rd	0 1 1 0 0 1 1	ROL
0 1 1 0 0 0 0	rs2	rs1	1 0 1	rd	0 1 1 0 0 1 1	ROR
0 1 1 0 0 0 0	shamt	rs1	1 0 1	rd	0 0 1 0 0 1 1	RORI
0 0 1 0 1 0 0	0 0 1 1 1	rs1	1 0 1	rd	0 0 1 0 0 1 1	ORC.B
0 1 1 0 1 0 0	1 1 0 0 0	rs1	1 0 1	rd	0 0 1 0 0 1 1	REV8

31–25	24–20	19–15	14–12	11–7	6–0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	rd	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11|19:12]				rd	opcode	J-type

Zri: "Load/Store indirect with Index" Extension

31–25	24–20	19–15	14–12	11–7	6–0	
0 0 0 0 0 0 0	rs2	rs1	1 1 1	rd	0 0 0 0 0 1 1	LB.R
0 0 0 0 0 0 1	rs2	rs1	1 1 1	rd	0 0 0 0 0 1 1	LH.R
0 0 0 0 0 1 0	rs2	rs1	1 1 1	rd	0 0 0 0 0 1 1	LW.R
1 0 0 0 0 0 0	rs2	rs1	1 1 1	rd	0 0 0 0 0 1 1	LBU.R
1 0 0 0 0 0 1	rs2	rs1	1 1 1	rd	0 0 0 0 0 1 1	LHU.R
0 0 0 0 0 0 0	rs3	rs1	1 1 1	rs2	0 1 0 0 0 1 1	SB.R
0 0 0 0 0 0 1	rs3	rs1	1 1 1	rs2	0 1 0 0 0 1 1	SH.R
0 0 0 0 0 1 0	rs3	rs1	1 1 1	rs2	0 1 0 0 0 1 1	SW.R

```
lb    rd, rs2(rs1)
lh    rd, rs2(rs1)
lw    rd, rs2(rs1)
lbu   rd, rs2(rs1)
lhu   rd, rs2(rs1)
sb    rs2, rs3(rs1)
sh    rs2, rs3(rs1)
sw    rs2, rs3(rs1)
```

31–25	24–20	19–15	14–12	11–7	6–0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	rd	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11|19:12]				rd	opcode	J-type

Zor: “Objective RISC” Extension

Unprivileged:

31–25	24–20	19–15	14–12	11–7	6–0		
0 0 0 0 0 0 0	rs2	rs1	0 0 0	rs3	0 0 0 1 0 1 1	SP.R	R
0 0 0 0 0 0 1	rs2	rs1	0 0 0	rd	0 0 0 1 0 1 1	LP.R	R
0 0 0 0 0 1 0	index[4:0]	frame	0 0 0	rs1	0 0 0 1 0 1 1	SV	R
0 0 0 0 0 1 1	index[4:0]	frame	0 0 0	rd	0 0 0 1 0 1 1	RST	R
0 0 0 0 1 0 0	zero	rs1	0 0 0	rd	0 0 0 1 0 1 1	QDTB	R
0 0 0 0 1 0 1	zero	rs1	0 0 0	rd	0 0 0 1 0 1 1	QDTH	R
0 0 0 0 1 1 0	zero	rs1	0 0 0	rd	0 0 0 1 0 1 1	QDTW	R
0 0 0 0 1 1 1	zero	rs1	0 0 0	rd	0 0 0 1 0 1 1	QDTD	R
0 0 0 1 0 0 0	zero	rs1	0 0 0	rd	0 0 0 1 0 1 1	QPI	R
0 0 0 1 0 0 1	zero	zero	0 0 0	rd	0 0 0 1 0 1 1	GCP	R
0 0 0 1 1 0 0	zero	frame	0 0 0	frame	0 0 0 1 0 1 1	POP	R
0 0 1 0 0 0 1	zero	zero	0 0 0	zero	0 0 0 1 0 1 1	RTLIB	R
0 0 1 0 0 1 0	zero	zero	0 0 0	zero	0 0 0 1 0 1 1	CPFC	R
0 0 1 0 0 1 1	zero	zero	0 0 0	zero	0 0 0 1 0 1 1	CHECK	R
imm[11:5]	rs2	rs1	0 0 1	imm[4:0]	0 0 0 1 0 1 1	SP	S
imm[11:0]		rs1	0 1 0	rd	0 0 0 1 0 1 1	LP	I
imm[11:0]		rs1	0 1 1	ra	0 0 0 1 0 1 1	JLIB	I
0 0 0 0 0 0 0	rs2	rs1	1 0 0	rd	0 0 0 1 0 1 1	ALC	R
pi[11:0]		rs1	1 0 1	rd	0 0 0 1 0 1 1	ALCI.P	I
dt[11:0]		rs1	1 1 0	rd	0 0 0 1 0 1 1	ALCI.D	I
dt[6:0]	0 0 0 0 0	rd	1 1 1	pi[4:0]	0 0 0 1 0 1 1	ALCI	S
dt[6:0]	0 0 0 1 0	frame	1 1 1	pi[4:0]	0 0 0 1 0 1 1	PUSHG	S
dt[6:0]	0 0 0 1 1	frame	1 1 1	pi[4:0]	0 0 0 1 0 1 1	PUSH	S

Machine Mode:

31–26	25	24–20	19–15	14–12	11–7	6–0		
1 1 1 1 1 1	0	0 0 0 0 0	0 0 0 0 0	0 0 0	rd	1 1 1 0 0 1 1	ALCB	R
1 1 1 1 1 1	1	rs2	rs1	0 0 0	rd	1 1 1 0 0 1 1	CIOP	R
1 1 1 1 1 1	0	1 0 0 0 0	rs1	0 0 0	rd	1 1 1 0 0 1 1	CCP	R
1 1 1 1 1 1	0	1 0 0 0 1	rs1	0 0 0	rd	1 1 1 0 0 1 1	RPR	R
1 1 1 1 1 1	0	1 0 1 0 0	rs1	0 0 0	rd	1 1 1 0 0 1 1	QPIR	R
1 1 1 1 1 1	0	1 0 1 0 1	rs1	0 0 0	rd	1 1 1 0 0 1 1	QDTR	R
1 1 1 1 1 1	0	1 0 1 1 0	rs1	0 0 0	rd	1 1 1 0 0 1 1	QPTR	R
1 1 1 1 1 1	0	0 0 0 0 0	0 0 0 1 0	0 0 0	rd	1 1 1 0 0 1 1	SEAL	R
1 1 1 1 1 1	0	0 0 0 0 0	0 0 0 1 1	0 0 0	rd	1 1 1 0 0 1 1	UNSL	R

Misc:

reg	alias	reg	alias
x0	zero	x16	a6
x1	ra ~~rix~~	x17	a7
x2	frame	x18	s2
x3	~~rcd~~/root/core	x19	s3
x4	ctxt	x20	s4
x5	t0	x21	s5
x6	t1	x22	s6
x7	t2	x23	s7
x8	s0	x24	s8
x9	s1	x25	s9
x10	a0	x26	s10/bm
x11	a1	x27	cnst
x12	a2	x28	t3
x13	a3	x29	t4
x14	a4	x30	t5
x15	a5	x31	t6

pseudo-instruction	implemented as
`lcp rd, imm(rs1)`	`lp rd, imm(rs1)` `sp x0, imm(rs1)`
`lcp.r rd, imm(rs1)`	`lp.r rd, rs2(rs1)` `sp.r x0, rs2(rs1)`
`scp rs2, imm(rs1)`	`sp rs2, imm(rs1)` `addi rs2, x0,0`
`scp.r rs2, rs3(rs1)`	`sp.r rs2, rs3(rs1)` `addi rs2, x0,0`
`pusht pi,dt`	`alci frame, pi,dt`

Implementation:

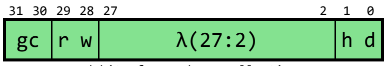
Instruction	**rdst**	**rdat**	**rptr**	**raux**	**imm**
sb/h/w	zero	ra.rix	rs1	rs2	imm
lb/bu/h/hu/w	rd	---	rs1	ra	imm
sp	zero	ra.rix	rs1	rs2	imm
lp	rd	---	rs1	ra	imm
sb/h/w.r	zero	rs3	rs1 (≠ frame)	rs2	---
lb/bu/h/hu/w.r	rd	rs2	rs1 (≠ frame)	---	---
sp.r	zero	rs3	rs1 (≠ frame)	rs2	---
lp.r	rd	rs2	rs1 (≠ frame)	---	---
sv	zero	ra.rix	frame	rs1	index
rst	rd	ra.rix	frame	bm	index
qdtx					
qpi					
gcp					
pop	frame	ra.rix	frame	---	---
jlib	ra	frame	rs1	ra	imm
jal	rd	frame	---	ra	imm
jr	rd	frame	rs1	ra	imm
rtlib	ra	ra.rix	ra	frame	---
alc	rd (≠ frame)	rs1	alc_params	rs2	---
alci.p	rd (≠ frame)	rs1	alc_params	---	pi
alci.d	rd (≠ frame)	rs1	alc_params	---	dt
alci	rd	ra.rix	alc_params	frame	pi & dt
pushg	rd	ra.rix	alc_params	frame	pi & dt
push	rd	ra.rix	alc_params	frame	pi & dt
alcb					
ciop	rd	rs1	---	rs2	---
rpr					
qpir					
qdtr					
qptr					
seal					
unsl					

<table>
<tr><th></th><th>31</th><th>30</th><th>29</th><th></th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>ra.rix</td><td>lib entry</td><td colspan="7">rix(30:1)</td><td>color</td></tr>
<tr><td>frame</td><td colspan="5">frame(31:3)</td><td>1</td><td>0</td><td>color</td></tr>
<tr><td>pi</td><td>uini</td><td colspan="5">pi(30:2)</td><td>bumper/gc</td><td>gc</td></tr>
<tr><td>dt</td><td>rc</td><td>ri</td><td colspan="6">dt(29:0)</td></tr>
</table>

instruction	condition	action
jlib	ra.rix(color) != frame(color) target ptr != ra.rcd	set ra.rix(lib entry), toggle rix(color)
jal ra, ... or jr ra, ...	ra.rix(color) != frame(color)	clear ra.rix(lib entry), toggle rix(color)
pushx	ra.rix(color) = frame(color)	toggle frame(color)
pop	ra.rix(color) != frame(color)	toggle frame(color)
jr ..., 0(ra)	ra.rix(color) = frame(color)	toggle ra.rix(color) if ra.rix(lib entry) = 1 do cross code-object return else stay in this code-object

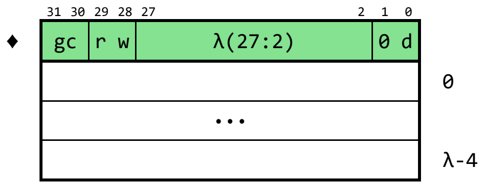
OBJECTS

Generic Header

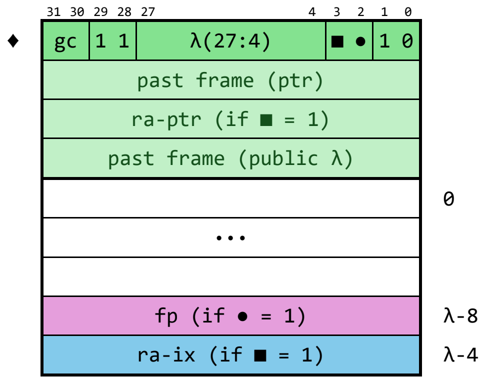



```
gc: reserved bits for garbage collection
r:  readable
w:  writable
λ:  length of this object
h:  extended header (header + pointer + ix + attr)
d:  data only (no pointers allowed)
```

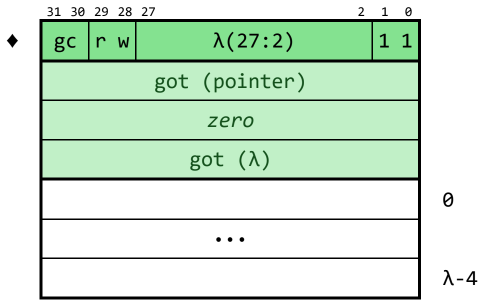
Ordinary



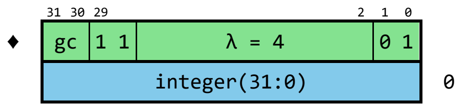
Frame



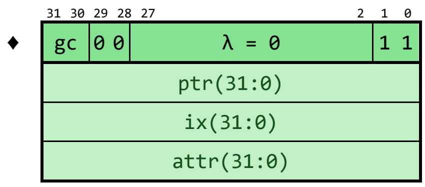
Executable



Immediate (Primitive)

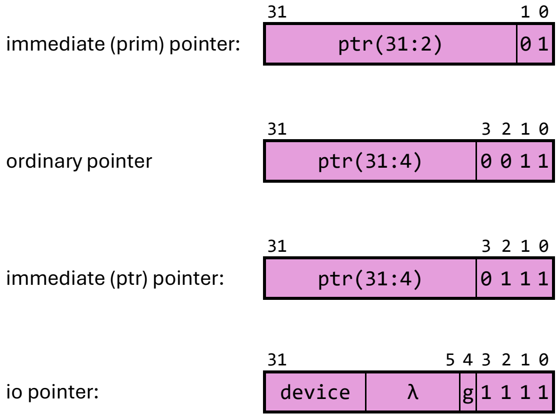


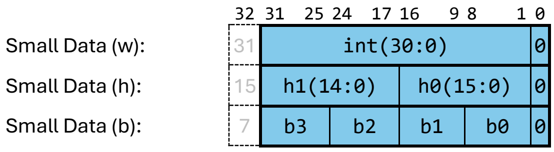
Immediate (Pointer)



POINTERS & DATA

(in memory)





Allocate immediate primitive if:

- sw and rs(30) ≠ rs(31)
- sh at h1 and rs(14) ≠ rs(15)
- sb at b3 and (rs(7) = 1 or rs < 0)

Invariant 1: There can never be a data-only object with an extended header which is not executable!!
Invariant 2: All objects with extended headers which are not data-only are implicitly frame objects!!

REGISTER FILE & PIPELINE

Architectural Registers (x0-x31, scratch, nano-scratch):

	T	31	4 3 2 1 0	31 0	Flags	25 2 1 0
data	0	value(31:0)		alc_addr	0 0 0 0	zero
ordinary pointer	1	ptr(31:4)	0 0 0 0	index(31:0)	r w x d	λ(27:2)
frame-type	1	ptr(31:4)	1 1 1 c	index(31:0)	1 1 0 0	λ(27:4) ■ ●
ra	1	ptr(31:4)	1 1 0 c	index(31:0)	r w 1 1	λ(27:2)
io pointer	1	ptr(31:4)	1 0 0 0	index(31:0)	r w 0 1	λ(27:2)
nano-scratch	0	value(31:0)				

Microarchitectural Registers:

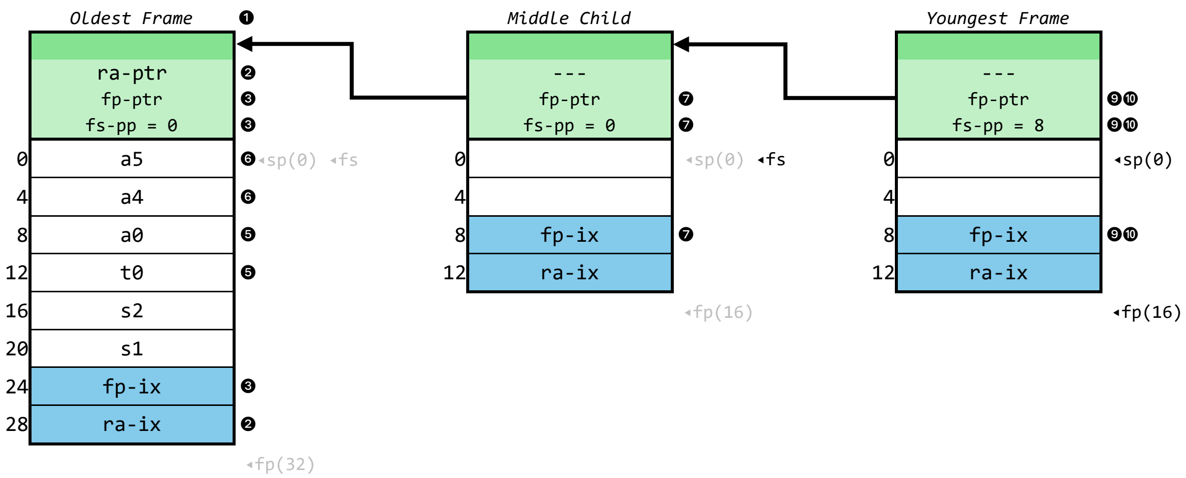
	T	31	4 3 2 1 0	31 0	Flags	25 0
got	1	ptr(31:0)			0 1 0 0	λ(27:2)
scratch (scr)						

FRAME OPERATIONS

```
❶  addi   sp, sp,-32
❷  sw     ra, 28(sp)
❸  sw     fp, 24(sp)
   sw     s1, 20(sp)
   sw     s2, 16(sp)
❹  addi   fp, sp,32
❺  sw     t0, -20(fp)
❺  sw     a0, -24(fp)
❻  sw     a4, 4(sp)
❻  sw     a5, 0(sp)
   jal    ra, label1
```

```
   addi   sp, sp,-16
   sw     ra, 12(sp)
❼  sw     fp, 8(sp)
   addi   fp, sp,16
❽  lw     t0, 0(fp)
❽  lw     t1, 4(fp)
   jal    ra, label2
```

```
   addi   sp, sp,-16
   sw     ra, 12(sp)
❾  sw     fp, 8(sp)
   addi   fp, sp,16
   ...
❿  lw     fp, 8(sp)
```



❶ allocate a new stack frame object via addi on sp

❷ store return index in designated “ra-spot” (aka. last element of frame). If ra was produced by a cross-object jump, ra-ptr is stored into the frame header.

❸ store index of frame pointer (fp/s0) into designated “fp-spot” (aka. second to last element of frame).
also stores the current value of fs (past-public) into the fp-pp spot.
also stores the base-pointer of the frame-object into the fp-ptr spot.
also writes fs(cp) to fs(pp), set fs(cp) = 0 and fs(ptr) = fp(ptr).

❹ overwrite fp with current stack frame pointer

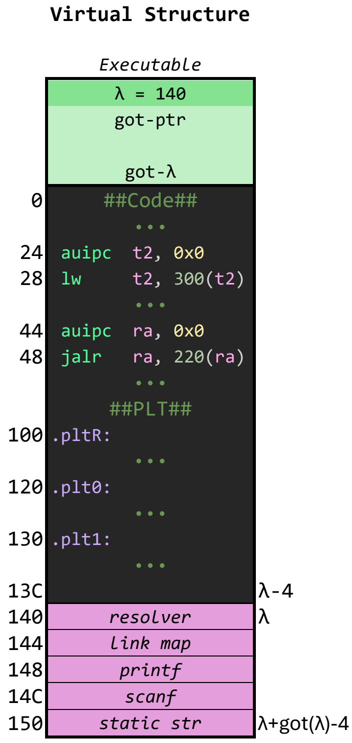
❺ sw-instructions using fp shorten the current-public-space. If the store address is lower than the current-public index, the index is decremented accordingly.

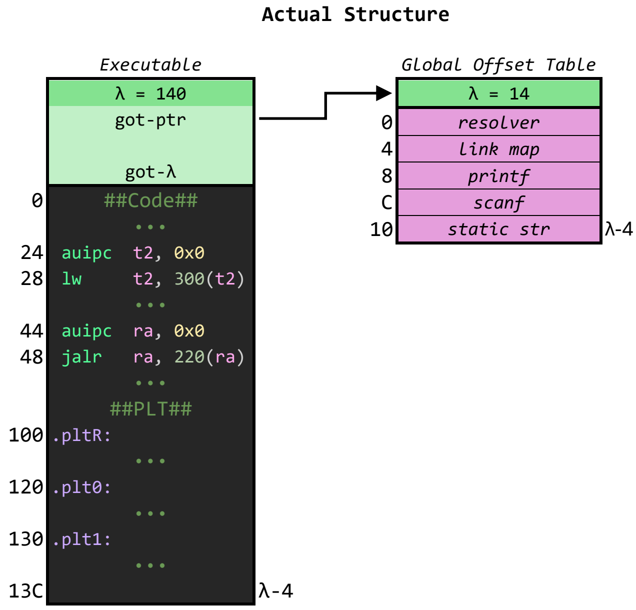
❻ sw-instructions using sp widen the current public-space. If the store address is higher than the current-public index <u>and</u> the store was executed using an argument register (a0-a7), the index is incremented accordingly.

❽ When an access via fp is detected, where the index would be too big for the current stack frame (fp(index)+offset > fp(λ)), then fs is used for the memory access instead. The index for the fs access calculates as: fs(offset) = fp(index)+offset-fp(λ).
if fs(offset) > fs(pp), an out of bounds exception is thrown.

❿ *How does fs-pp get restored?*

CODE SEGMENTATION





User Mode Instructions

Instruction	rd	rs1	rs2	cr	imm	Decoder Decision
lui	rd	---	---	-	imm	
auipc	rd	---	---	-	imm	
jal	rd	---	sp	●	imm	
jalr	*rd*	*rs1*	*---*	*-*	*imm*	
A jalr	rd	rs1	sp	●	imm	*always*
A lgt	got	rs1	---	-	---	*always (instead of nop)*
bcc	---	rs1	rs2	-	imm	
lb/bu/h/hu/w	*rd*	*rs1*	*---*	*-*	*---*	
A lb/bu/h/hu/w	rd	rs1	fs	●	---	*if rs1 = fp or rs1 = sp*
B lb/bu/h/hu/w	rd	rs1	got	●	---	*otherwise*
sb/h/w	*---*	*rs1*	*rs2*	*-*	*imm*	
A loadmux	scr	rs1	rs2	●	imm	*if sb and imm(0) = 1*
A1 sb_m/h_m	---	rs1	scr	●	imm	*or sh and imm(1) = 1* *WRONG*
A2 sb_m/h_m	fs	rs1	scr	●	imm	*if rs1 = sp*
B sb/h/w	fs	rs1	rs2	●	imm	*if rs1 = sp*
C sb/h/w	---	rs1	rs2	●	imm	*otherwise*
addi	*rd*	*rs1*	*---*	*-*	*imm*	
A push	sp	sp	---	●	imm	*if rd = sp and rs1 = sp and imm > 0*
B pop	sp	sp	---	-	imm	*if rd = sp and rs1 = sp and imm < 0*
C setpublic	zero	sp	---	●	imm	*if rd = zero and rs1 = sp and imm ≥ 0*
D addi	rd	rs1	---	-	imm	*otherwise*
arithi	rd	rs1	---	-	imm	
arith	rd	rs1	rs2	-	---	
alc	rd	rs1	alc_params	-	---	
alci	rd	---	alc_params	-	imm	
alc.d	rd	rs1	alc_params	-	---	
alci.d	rd	---	alc_params	-	imm	
qsz	rd	rs1	---	-	---	

Supervisor Mode Instructions:

Instruction	rd	rs1	rs2	cr	imm	Notes
sb/h/w.r	---	rs1	rs2	-	imm	*“store raw”, allows stores at any point in memory. Uses rs1 as base-ptr*
lb/bu/h/hu/w.r	rd	rs1	---	-	---	*“load raw”, same as store raw*
dtp	rd	rs1	---	-	---	*“data to pointer”, creates a pointer from data*
ptd	rd	rs1	---	-	---	*“pointer to data”, extracts base address of pointer as data*
itd	rd	rs1	---	-	---	*“index to data”, extracts index of pointer as data*

Problem: we only know if we need to box an immediate in execute. How do we handle instructions, which split into multiple nano-instructions in execute?

DOKUMENTATION: ELF-FILES

“Executable and Linkable Format”-Files bestehen mindestens aus einem Header, einer “Program Header Table” und einer “Section Header Table”. Im Header werden Informationen über das ELF-File selbst gespeichert, wie z.B. die Prozessorarchitektur, für welche das Programm kompiliert wurde und die Positionen der PHT und der SHT in Relation zum File-Anfang. In einem Program Header werden Informationen gespeichert, die dem Betriebssystem angeben, wie viele und welche Arten von virtuellen Seiten für dieses Programm benötigt werden. In einem Section Header wird angegeben, in welche Einzelteile das Programm zerlegt wurde und ob noch mehr Informationen über das Programm im ELF-File zu finden sind (z.B. für relocatable Programme).

Daten

Statische Daten werden von einem Compiler über Assemblerdirektiven immer so in die `.data` bzw. `.rodata` Sektionen abgelegt, sodass sie in der Symboltabelle des ELF-Files immer als Objekt mit seiner Größe eindeutig erkennbar sind.

```
//C-Code                                                //C-Code
static char stringA[] = "hello world!";                 static const char stringB[] = "hello world!";


#Resultierender Assembly-Code                           #Resultierender Assembly-Code
.data                                                   .rodata

.type    stringA, @object                               .type    stringB, @object
stringA: .asciz "hello world!"                          stringB: .asciz "hello world!"
.size    stringA, .-stringA                             .size    stringB, .-stringB


//Section Header Table im erzeugten ELF-File
Section Headers:
  [Nr] Name      Type      Address   Offset    Size      EntSize   Flags Link Info Align
       ...
  [ 5] .data     PROGBITS  00002010  000003b4  0000000d  00000000   WA     0    0    4
  [ 6] .rodata   PROGBITS  00002020  000003c4  0000000d  00000000    A     0    0    4
       ...

//Symbol Table im erzeugten ELF-File
Symbol table '.symtab' contains 60 entries:
   Num:  Value      Size Type    Bind    Vis       Ndx Name
       ...
    49: 00000000    13 OBJECT  LOCAL   DEFAULT    5 stringA
    50: 00000000    13 OBJECT  LOCAL   DEFAULT    6 stringB
       ...
```

Ein Zugriff auf solche statischen Daten kann in executables und muss in relocatables über die Global Offset Table (GOT) stattfinden. Angenommen ein Programm läge an der physikalischen Adresse 0x0 und seine zugehörige GOT an der Adresse 0x1000 und am Offset 8 der GOT stünde die Adresse für das Symbol stringA, dann würde mit folgenden Assembly befehlen auf diesen Eintrag zugegriffen werden.

```
    auipc   t2, 0x1    # R_RISCV_GOT_HI20 (symbol), R_RISCV_RELAX
    lw      t2, 8(t2)  # R_RISCV_PCREL_LO12_I (auipc), R_RISCV_RELAX
```

In einer executable können die Immediates für diese Befehlssequenz direkt befüllt werden, da der Abstand des Programms zur GOT schon beim Kompilieren des Programms bekannt ist. Bei einem relocatable Programm belässt der Compiler diese Immediates mit 0 und markiert die Befehle in der „Relocation Section“ als unaufgelöst. Sowohl die GOT als auch die .data oder .rodata Sektionen können vom Betriebssystem beim Laden des Programms an beliebige Stellen im Speicher platziert werden. Sind alle Sektionen platziert, kann der Dynamische Linker anhand der Tags der Einträge in der Relocation Section herausfinden, wie er die Immediates für die aufzulösenden Symbole zu berechnen hat. `R_RISCV_GOT_HI20` z.B. bedeutet, dass für diese Instruktion die obersten 20 Bits der Differenz aus Position der Instruktion und Position der GOT benötigt. Die Relax Tags sollen anzeigen, dass es je nach Positionierung möglich sein könnte, eine der beiden Instruktionen zu sparen falls z.B. Instruktion und GOT nah genug beieinander liegen.

Code

Bla bla bla Procedure Linkage Table

```
#PROCEDURE LINKAGE TABLE#
00000080 <.plt>:
.plt
.pltR:  auipc   t2, %pcrel_hi(.got.plt)
        sub     t1, t1, t3                # t1 = difference between caller and .pltR + 12
        lw      t3, %pcrel_lo(.pltR)(t2)  # t3 = addr(_dl_runtime_resolve)
        addi    t1, t1, -44               # subtract size of .pltR (32) and jalr offset in caller (12)
        addi    t0, t2, %pcrel_lo(.pltR)  # t0 = start of .got
        srli    t1, t1, 2                 # index of .plt entry in .got.plt
        lw      t0, 4(t0)                 # link map
        jr      t3

.plt0:  auipc   t3, %pcrel_hi(functionA@.got.plt)
        lw      t3, %pcrel_lo(.plt0)(t3)
        jalr    t1, t3
        nop
.plt1:  auipc   t3, %pcrel_hi(functionB@.got.plt)
        lw      t3, %pcrel_lo(.plt1)(t3)
        jalr    t1, t3
        nop
.plt2:  ...

#GLOBAL OFFSET TABLE#
000010ac <.got.plt>:
.got.plt
         .word 0xffffffff  #to be filled with address of the dynamic resolver
         .word 0x00000000  #to be filled with pointer to “link map”
         .word 0x00000080  #func entry 0
         .word 0x00000080  #func entry 1
         .word 0x00000080  #func entry 2
         ...
```