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 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

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 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

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

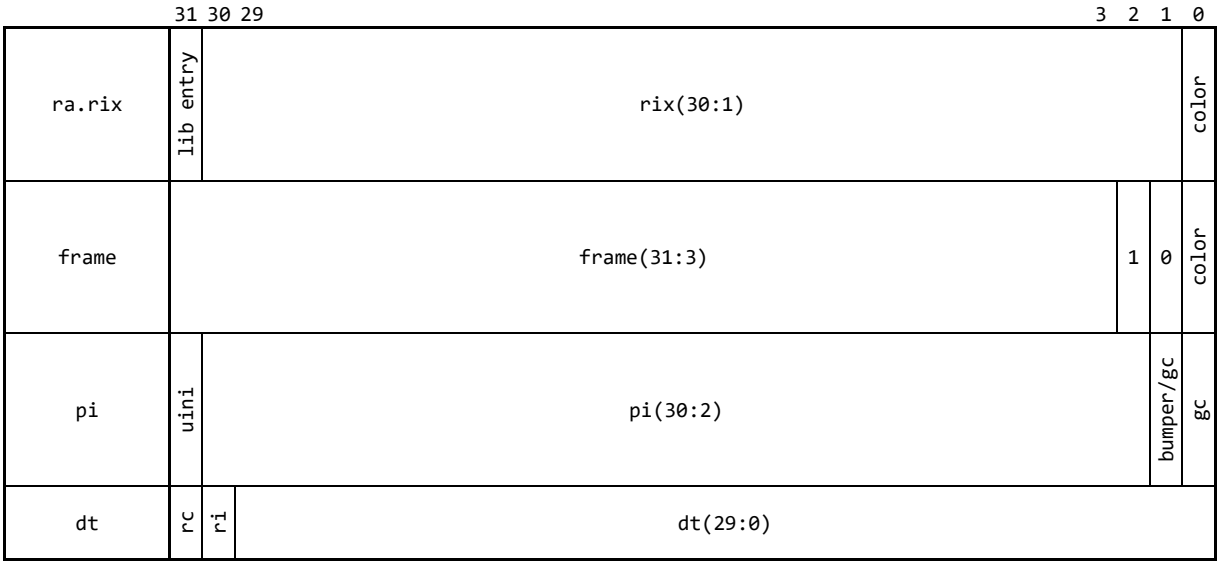
Misc:

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:

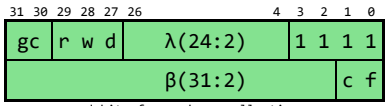
Instruction	rdst	rdat	rptr	raux	imm
sb/h/w	zero	ra.rix	rs1	rs2	imm
lb/bu/h/hu/w	rd	---	rs1	ra	imm
sp	zero	ra.rix	rs1	rs2	imm
lp	rd	---	rs1	ra	imm
sb/h/w.r	zero	rs3	rs1 (≠ frame)	rs2	---
lb/bu/h/hu/w.r	rd	rs2	rs1 (≠ frame)	---	---
sp.r	zero	rs3	rs1 (≠ frame)	rs2	---
lp.r	rd	rs2	rs1 (≠ frame)	---	---
sv	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					



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

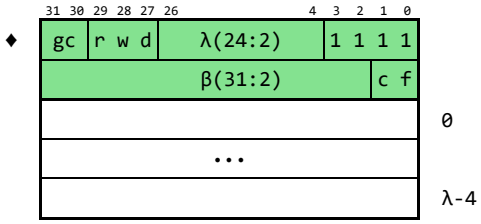
OBJECTS

Generic Header

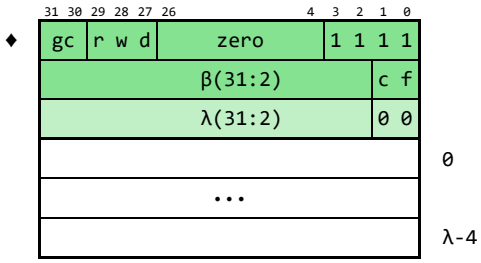


gc: reserved bits for garbage collection
r: readable
w: writable
d: data only (no pointers allowed)
λ: length of this object
β: current access barrier
f: is stack frame object?
c: color of stack frame if f = 1, else don't care

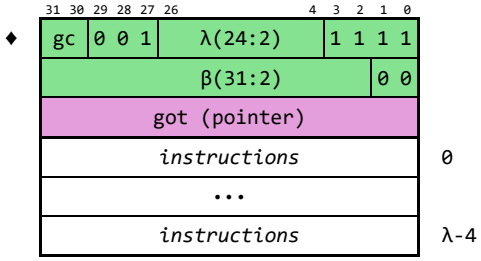
Ordinary



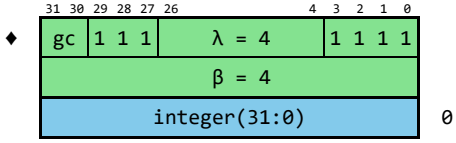
Long



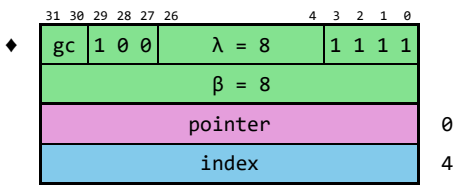
Executable



Immediate (Primitive)



Immediate (Pointer)

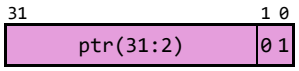


Invariant 1: data-only objects which are not readable and not writable are implicitly executable
Invariant 2: Objects with λ = 0 in their base-header have λ moved to the address following β

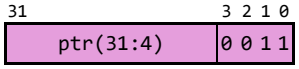
POINTERS & DATA

(in memory)

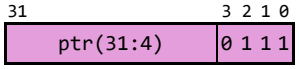
immediate (prim) pointer:



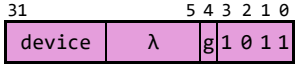
ordinary pointer

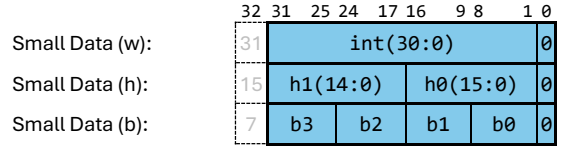


immediate (ptr) pointer:



io pointer:



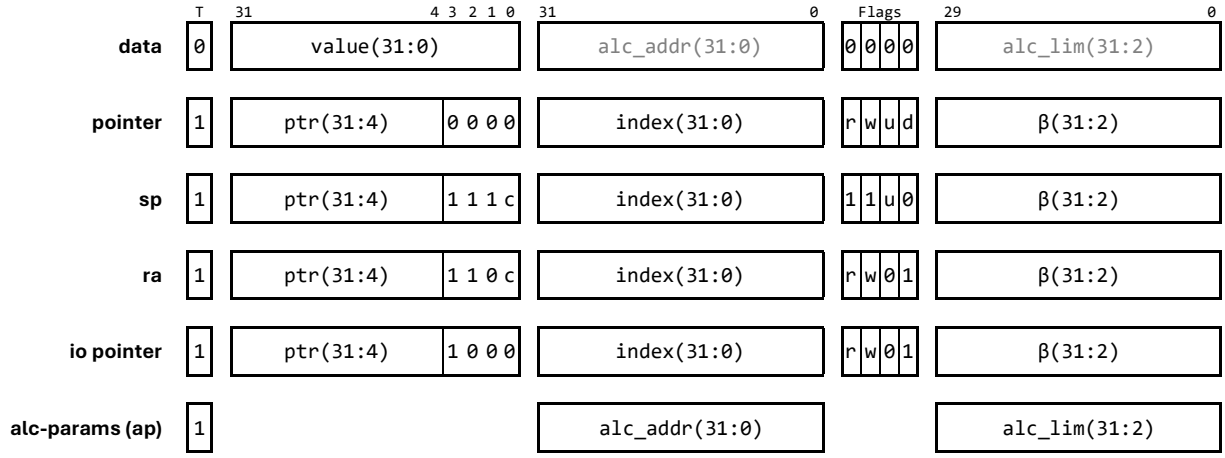


Allocate immediate primitive if:

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

REGISTER FILE & PIPELINE

Architectural Registers (x0-x31, alc-params):



Tags:
r read access, **w** write access, **u** 0 when β = λ else 1, **d** data only

Microarchitectural Registers:

Do this with gp?

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

Zor: “Objective RISC” Extension 2

Unprivileged:

31 25	24 20	19 15	14 12	11 7	6 0		
0 0 0 0 0 0 0	zero	rs1	0 0 0	rd	0 0 0 1 0 1 1	ALC	R
size[13:2]		sp	0 1 0	rd	0 0 0 1 0 1 1	ALCI	I
0 0 0 0 0 0 0	zero	rs1	0 0 1	rd	0 0 0 1 0 1 1	ALC.D	R
size[13:2]		sp	0 1 1	rd	0 0 0 1 0 1 1	ALCI.D	I
0 0 0 0 0 0 0	zero	rs1	1 0 0	rd	0 0 0 1 0 1 1	QSZ	R
0 0 0 0 0 0 0	zero	rs1	1 0 1	rd	0 0 0 1 0 1 1	LEB	R
0 0 0 0 0 0 0	zero	rs1	1 1 0	zero	0 0 0 1 0 1 1	CLR	R

Machine Mode:

31 26	25	24 20	19 15	14 12	11 7	6 0		
0 1 1 1 1 1	1	rs2	rs1	0 0 0	rd	1 1 1 0 0 1 1	DTP	R
1 0 1 1 1 1	1	zero	rs1	0 0 0	rd	1 1 1 0 0 1 1	PTD	R
1 1 0 1 1 1	1	zero	rs1	0 0 0	rd	1 1 1 0 0 1 1	ITD	R

Misc:

reg	alias	reg	alias
x0	zero	x16	a6
x1	ra	x17	a7
x2	sp	x18	s2
x3	gp (got)	x19	s3
x4	tp	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
x11	a1	x27	s11
x12	a2	x28	t3
x13	a3	x29	t4
x14	a4	x30	t5
x15	a5	x31	t6

pseudo-instruction	implemented as
push imm	alci sp, imm
pop	lw/d sp, 4(sp)

FRAME OPERATIONS

```c
void routine0(){
  int a = 8;
  int b[4];
  routine1(..., 3, b);
  ...
}
```

```c
void routine1(..., int x, int* arr){
  routine2(arr[0], arr[x]);
  ...
}
```

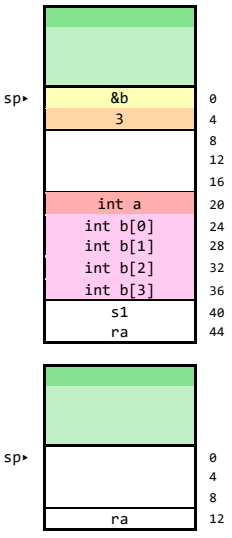
```c
int routine2(int q, int p){
  return q+p;
}
```

We support 4 different ways to use the stack:

Standard RISC-V ABI

access to the past stack frame allowed via sp and fp

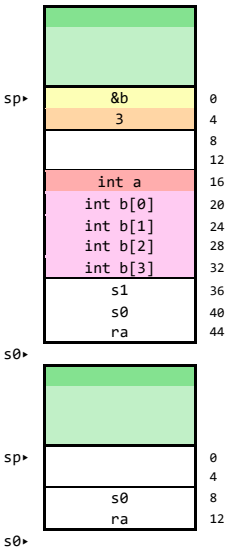
```
routine0:
 addi    sp, sp,-48
 sw      ra, 44(sp)
 sw      s1, 40(sp)

 li      t0, 8
 sw      t0, 16(sp)

 li      a7, 3
 sw      a7, 4(sp)
 addi    a7, sp,24
 sw      a7, 0(sp)
 jal     ra, routine1

routine1:
 addi    sp, sp,-16
 sw      ra, 12(sp)

 lw      a1, 16(sp)
 lw      a0, 0(a1)
 lw      t0, 20(sp)
 add     a1, a1,t0
 lw      a1, 0(a1)
 jal     ra, routine2

routine2:
 add     a0, a0,a1
 ret
```



Standard RISC-V ABI (-fno-omit-frame-pointer)

access to the past stack frame allowed only via fp

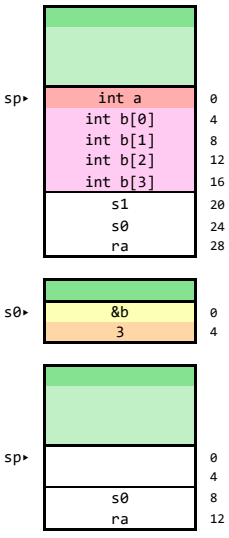
```
routine0:
 addi    sp, sp,-48
 sw      ra, 44(sp)
 sw      s0, 40(sp)
 addi    s0, sp,48
 sw      s1, 36(sp)

 li      t0, 8
 sw      t0, -32(s0)

 li      a7, 3
 sw      a7, 4(sp)
 addi    a7, s0,-28
 sw      a7, 0(sp)
 jal     ra, routine1

routine1:
 addi    sp, sp,-16
 sw      ra, 12(sp)
 sw      s0, 8(sp)
 addi    s0, sp,16

 lw      a1, 0(s0)
 lw      a0, 0(a1)
 lw      t0, 4(s0)
 add     a1, a1,t0
 lw      a1, 0(a1)
 jal     ra, routine2

routine2:
 add     a0, a0,a1
 ret
```



use fp for parameters

access to the past stack frame only with explicit ptr (not sp)

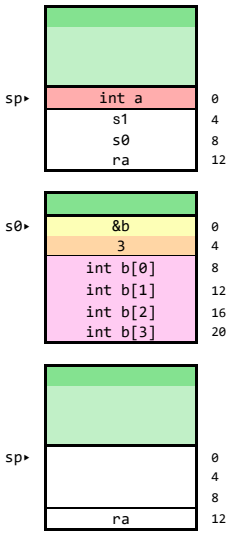
```
routine0:
 addi    sp, sp,-32
 sw      ra, 28(sp)
 sw      s0, 24(sp)
 sw      s1, 20(sp)

 li      t0, 8
 sw      t0, 0(sp)

 alci    s0, 8
 li      a7, 3
 sw      a7, 4(s0)
 addi    a7, sp,4
 sw      a7, 0(s0)
 jal     ra, routine1

routine1:
 addi    sp, sp,-16
 sw      ra, 12(sp)
 sw      s0, 8(sp)
 addi    s0, sp,16

 lw      a1, 0(s0)
 lw      a0, 0(a1)
 lw      t0, 4(s0)
 add     a1, a1,t0
 lw      a1, 0(a1)
 jal     ra, routine2

routine2:
 add     a0, a0,a1
 ret
```



split frame in private and public μ-frames

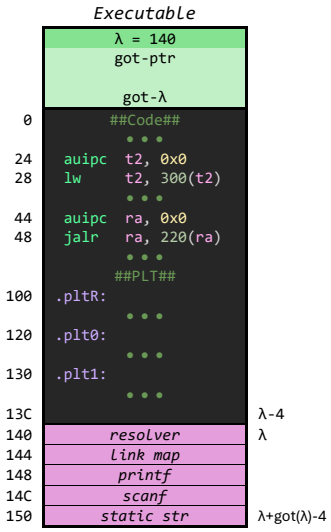
access to the past stack frame only with explicit ptr (not sp)

```
routine0:
 addi    sp, sp,-16
 sw      ra, 12(sp)
 sw      s0, 8(sp)
 alci    s0, 20
 sw      s1, 4(sp)

 li      t0, 8
 sw      t0, 0(sp)

 li      a7, 3
 sw      a7, 4(s0)
 addi    a7, s0,8
 sw      a7, 0(s0)
 jal     ra, routine1

routine1:
 addi    sp, sp,-16
 sw      ra, 12(sp)

 lw      a1, 0(s0)
 lw      a0, 0(a1)
 lw      t0, 4(s0)
 add     a1, a1,t0
 lw      a1, 0(a1)
 jal     ra, routine2

routine2:
 add     a0, a0,a1
 ret
```

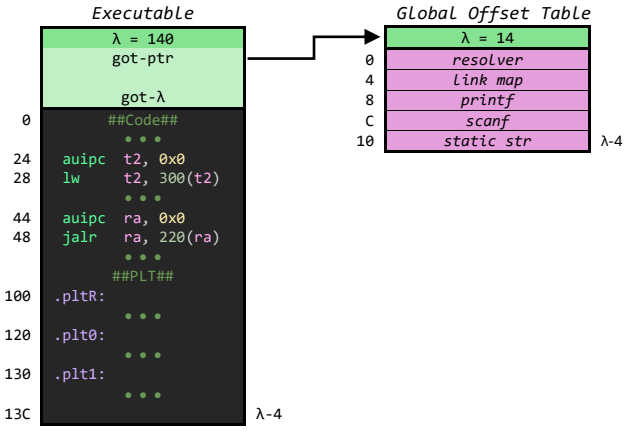


CODE SEGMENTATION

Virtual Structure



Actual Structure



User Mode Instructions (Single Cycle)

Instruction	rd	rs1	rs2	cr	imm	Decision
lui	rd	---	---	-	imm	
auipc	rd	---	---	-	imm	
jal	rd	---	sp	●	imm	
bcc	---	rs1	rs2	-	imm	
arithi	rd	rs1	---	-	imm	
arith	rd	rs1	rs2	-	---	
lb/bu/h/hu/w	rd	rs1	got	●	imm	
sb/h/w						*impossible as single-cycle*
alc	rd	rs1	ap	-	---	
alci	rd	sp	ap	●	imm	
alc.d	rd	rs1	ap	-	---	
alci.d	rd	sp	ap	●	imm	*(Frage: macht ein data-only frame eig. Sinn?)*
qsz	rd	rs1	---	-	---	
clr	rs1	rs1	---	-	imm	
leb	rd	rs1	---	-	---	

User Mode Instructions (Multi Cycle)

Instruction	rd	rs1	rs2	cr	imm	Decision
jalr	*rd*	*rs1*	*---*	*-*	*imm*	
A jalr	rd	rs1	sp	●	imm	*always*
A lgt	got	rs1	---	-	---	*always (instead of nop)*
sb/h/w	*---*	*rs1*	*rs2*	*-*	*imm*	
a alci.d	rs1	rs2	ap	●	4	*if rs2 is data and does not fit in word*
b alci	rs1	rs2	ap	●	8	*if rs2 is pointer with index ≠ 0*
A/B sb/h/w	rs1	rs1	rs2	●	imm	*always*

Supervisor Mode Instructions:

Instruction	rd	rs1	rs2	cr	imm	Notes
sb/h/w.x	---	rs1	rs2	-	imm	*“store raw”, allows stores at any point in memory. Uses rs1 as base-ptr*
lb/bu/h/hu/w.x	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*

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
static char stringA[] = "hello world!";


#Resultierender Assembly-Code
.data

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


//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
       ...
```

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


#Resultierender Assembly-Code
.rodata

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