

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	0	rs2	rs1	1	1	1	rd	0	1	1	0	0	1	1	ANDN				
0	1	0	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	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	0	rd	0	0	1	0	0	1	1	CLZ				
0	1	1	0	0	0	0	0	0	0	0	0	1	rs1	0	0	1	rd	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	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	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	rs1	1	0	0	rd	0	1	1	0	0	1	1	ZEXT.H				
0	1	1	0	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	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	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	0	rs1	1	0	1	rd	0	0	1	0	0	1	1	REV8

- R-type
- I-type
- S-type
- B-type
- U-type
- J-type

Zor: “Objective RISC” Extension 2

Unprivileged:

ALC	R
ALCI	I
ALC.D	R
ALCI.D	I
QSZ	R

Machine Mode:

DTP	R
PTD	R
ITD	R
LW.X	R
SW.X	R

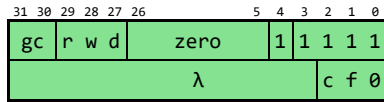
Misc:

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

[illegible]

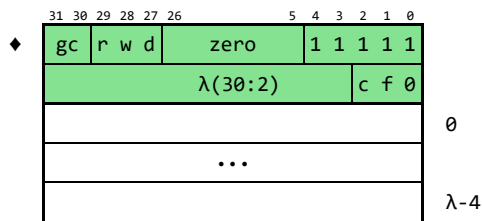
OBJECTS

Generic Header

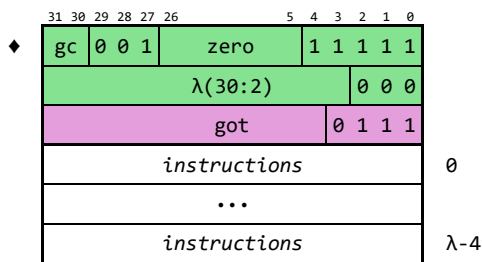


gc: reserved bits for garbage collection
 r: readable
 w: writable
 d: data only (no pointers allowed)
 λ: length of this object
 f: is stack frame object?
 c: color of stack frame if f = 1, else don't care

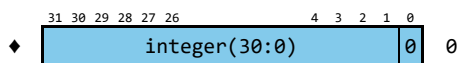
Ordinary



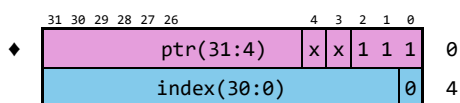
Executable



Immediate (Primitive)

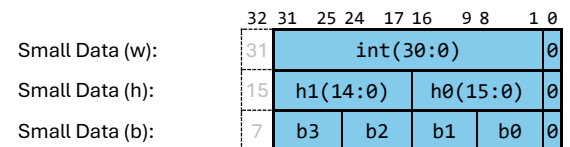
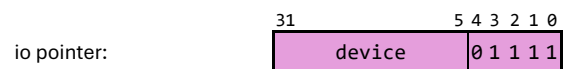
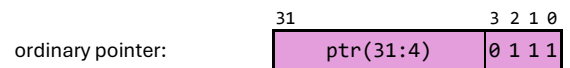
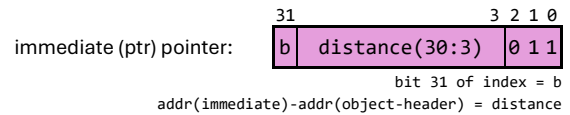
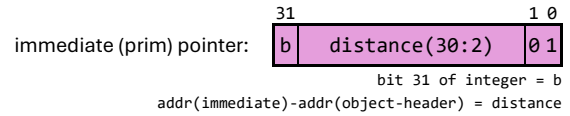


Immediate (Pointer)



POINTERS & DATA

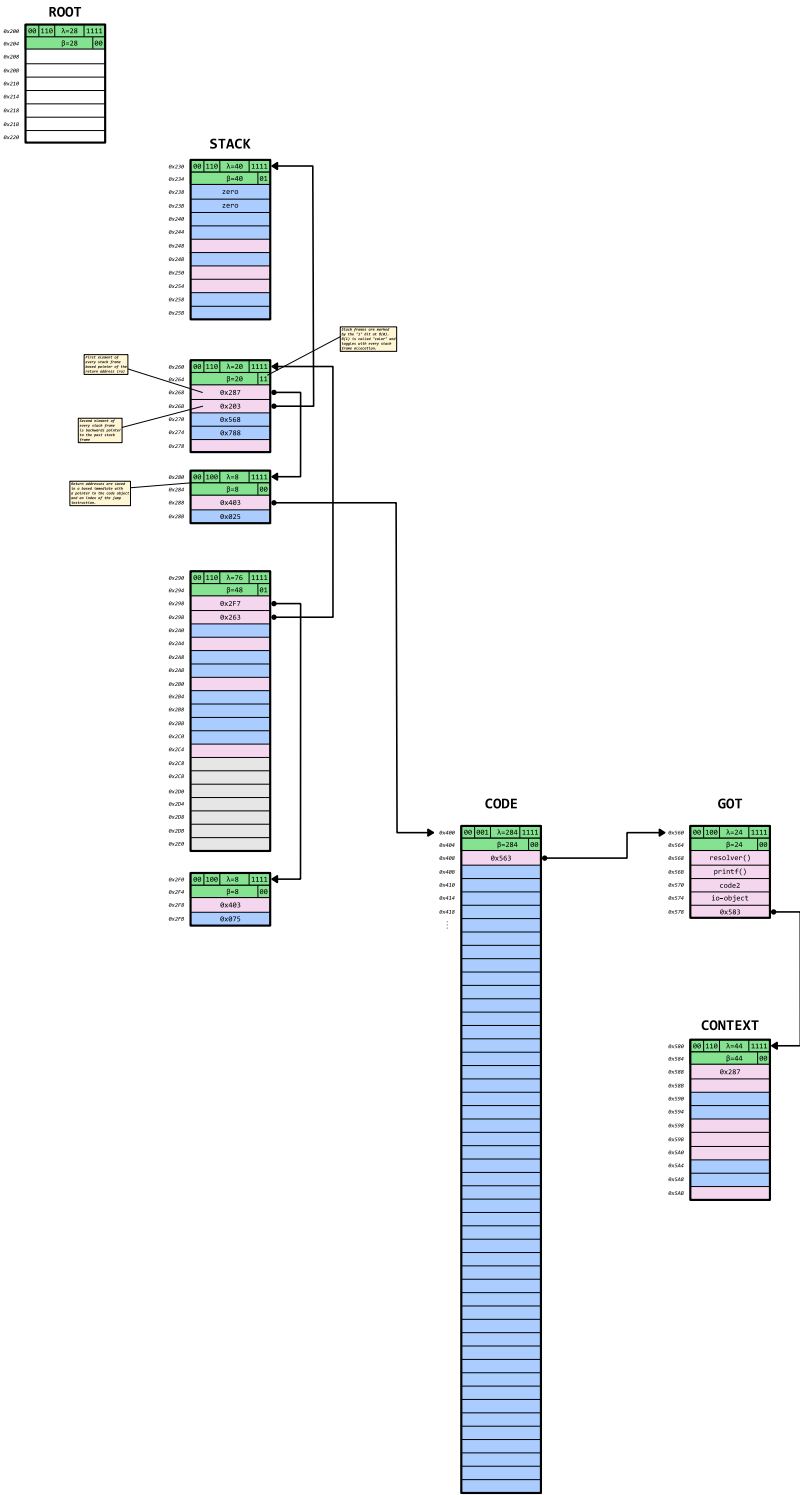
(inside objects)



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: data-only objects which are not readable and not writable are implicitly executable
Invariant 3: immediates can only be stored in 2^{29} -byte distance of its parent object-header



REGISTER FILE & PIPELINE

Architectural Registers (x0-x31):

	T	31	4	3	2	1	0	30	0	29	0
data	0	value(31:0)						zero		zero	
pointer	1	ptr(31:4)				r w d 1		index(30:0)		λ(31:2)	
sp	1	ptr(31:4)				1 1 0 c		index(30:0)		λ(31:2)	
ra	1	ptr(31:4)				r w 1 c		index(30:0)		λ(31:2)	
gp	1	ptr(31:4)				1 0 0 1		zero!		λ(31:2)	
io pointer	1	device				1 1 1 1		index(30:0)		λ(31:2)	

Flags/Tags:
r read access, w write access, d data only

Microarchitectural Registers:

	T	31	4	3	2	1	0	31	0	Flags	29	0
alc-params (ap)		ptr(31:0)										

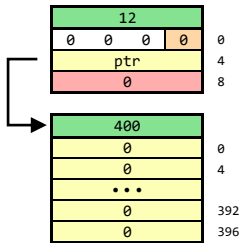
OBJECT INITIALIZATION

Deprecated yet again ;-;

CLEAR ON ALC

```
struct foo {  
    bool a = false;  
    int array[100];  
    int c = 0;  
} bar;
```

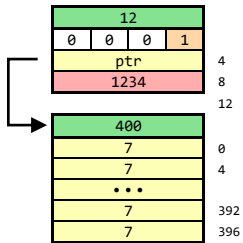
```
routine0:  
    alci    s0, 12  
.init0:  
    clr     s0  
    leb     t0, s0  
    beq     zero, t0, .init0  
    alci    t1, 400  
.init1:  
    clr     t1, zero  
    leb     t0, t1  
    beq     zero, t0, .init1  
    sw      t1, 4(s0)
```



INIT WITH DEFAULTS

```
struct foo {  
    bool a = true;  
    int array[100];  
    int c = 1234;  
} bar;  
...  
//forall  
    bar->array[i] = 7;
```

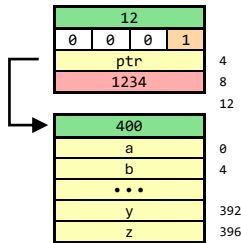
```
routine0:  
    alci    s0, 12  
    li      t1, 1  
    sb      t1, 0(s0)  
    alci    t2, 400  
    sw      t2, 4(s0)  
    li      t3, 1234  
    sw      t3, 8(s0)  
...  
    li      t4, 7  
.init1:  
    clr     t2, t4  
    leb     t5, t2  
    beq     zero, t5, .init1
```



DISALLOWED LAZY INIT

```
struct foo {  
    bool a = true;  
    int array[100];  
    int c = 1234;  
} bar;  
...  
bar->array = routine1();
```

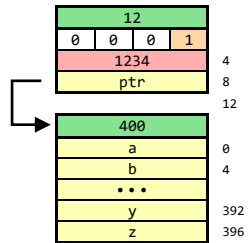
```
routine0:  
    alci    s0, 12  
    li      t1, 1  
    sb      t1, 0(s0)  
    li      t1, 1234  
    sw      t1, 8(s0)  
...  
    jal     ra, routine1  
    sw      a0, 4(s0)
```



ALLOWED LAZY INIT

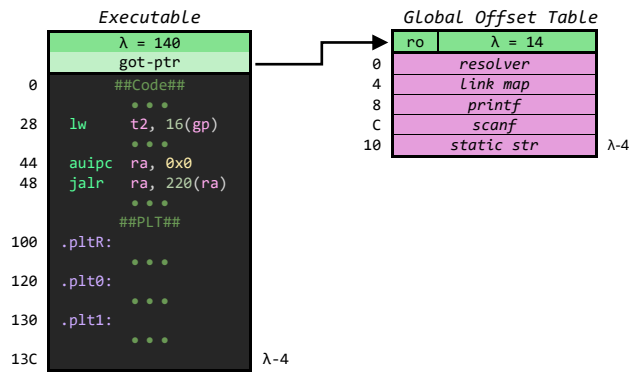
```
struct foo {  
    bool a = true;  
    int array[100];  
    int c = 1234;  
} bar;  
...  
bar->array = routine1();
```

```
routine0:  
    alci    s0, 12  
    li      t1, 1  
    sb      t1, 0(s0)  
    li      t1, 1234  
    sw      t1, 4(s0)  
...  
    jal     ra, routine1  
    sw      a0, 8(s0)
```



CODE SEGMENTATION

Executable-GOT linking



Code-Objects can only be exited via the procedure linkage table (plt) which uses entries of the global offset table (got). Upon creation of the code-object or on the first try to exit the code object, the supervisor puts the pointer to the target code-object into the global offset table, if (and only if) the source code-object is allowed to jump to that target.

User Mode Instructions (Single Cycle)

Instruction	rd	rs1	rs2	cr	imm	Notes
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	rd	rs1	---	●	imm	
sb/h/w	---	rs1	rs2	●	imm	
alc	rd	rs1	sp	●	---	if rd = sp, also stores sp to index 4
alci	rd	---	sp	●	imm	if rd = sp, also stores sp to index 4
alc.d	rd	rs1	---	-	---	zero, ra, gp and sp forbidden for rd
alci.d	rd	---	---	-	imm	zero, ra, gp and sp forbidden for rd
qsz	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)
lw	---	rs1	rs2	-	imm	
A lw	rd	rs1	---	●	imm	always
a lw	rd	rs1	---	●	imm	needed if rdata is immediate pointer

Machine Mode Instructions:

Instruction	rd	rs1	rs2	cr	imm	Notes
dtb	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
lw.x	rd	rs1	---	-	---	Load data from linear memory
sw.x	rd	rs1	rs2	-	---	store data to linear memory

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!";

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

#Resultierender Assembly-Code
.data

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

#Resultierender Assembly-Code
.rodata

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