1. Přípravná cvičení

```
0x00001000 in ?? ()
=> 0x00001000:  97 02 00 00     auipc   t0,0x0
1: /z $x5 = 0x00000000
2: /z $x6 = 0x00000000
3: /z $x7 = 0x00000000

Breakpoint 1, _start () at test.s:12
12              li x6, 1                # x6 = 1
=> 0x80000000 <_start+0>:       13 03 10 00     li      t1,1
1: /z $x5 = 0x80000000
2: /z $x6 = 0x00000000
3: /z $x7 = 0x00000000
(gdb) si
13              li x7, -2               # x7 = -2
=> 0x80000004 <_start+4>:       93 03 e0 ff     li      t2,-2
1: /z $x5 = 0x80000000
2: /z $x6 = 0x00000001
3: /z $x7 = 0x00000000
(gdb)
14              add x5, x6, x7          # x5 = x6 + x7
=> 0x80000008 <_start+8>:       b3 02 73 00     add     t0,t1,t2
1: /z $x5 = 0x80000000
2: /z $x6 = 0x00000001
3: /z $x7 = 0xfffffffe
(gdb)
stop () at test.s:17
17              j stop                  # Infinite loop to stop execution
=> 0x8000000c <stop+0>: 6f 00 00 00     j       0x8000000c <stop>
1: /z $x5 = 0xffffffff
2: /z $x6 = 0x00000001
3: /z $x7 = 0xfffffffe
(gdb)
```


Sečtení záporných čísel ponechaných v předchozím článku ukazuje, že tomu tak je v registru, x7 je -2 a x5 je -1.

Tady není moc co říkat, jen si projděte koncept doplňkového kódu.

Podívejme se na odčítání: sub

```
# Substract
# Format:
#   SUB RD, RS1, RS2
# Description:
#   The contents of RS2 is subtracted from the contents of RS1 and the result
#   is placed in RD.
```

```
    .text           # Define beginning of text section
.global _start      # Define entry _start

_start:
    li x6, -1       # x6 = -1
    li x7, -2       # x7 = -2
    sub x5, x6, x7      # x5 = x6 - x7

stop:
    j stop          # Infinite loop to stop execution

    .end            # End of file
```

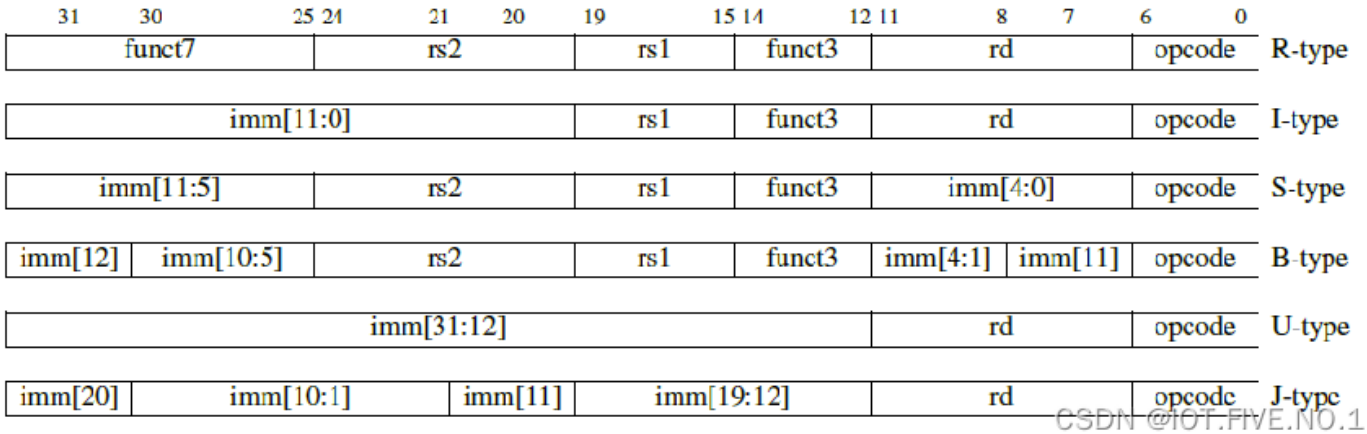
```
Reading symbols from test.elf...
Breakpoint 1 at 0x80000000: file test.s, line 12.
0x00001000 in ?? ()
=> 0x00001000:  97 02 00 00     auipc   t0,0x0
1: /z $x5 = 0x00000000
2: /z $x6 = 0x00000000
3: /z $x7 = 0x00000000

Breakpoint 1, _start () at test.s:12
12              li x6, -1               # x6 = -1
=> 0x80000000 <_start+0>:       13 03 f0 ff     li      t1,-1
1: /z $x5 = 0x80000000
2: /z $x6 = 0x00000000
3: /z $x7 = 0x00000000
(gdb) si
13              li x7, -2               # x7 = -2
=> 0x80000004 <_start+4>:       93 03 e0 ff     li      t2,-2
1: /z $x5 = 0x80000000
2: /z $x6 = 0xffffffff
3: /z $x7 = 0x00000000
(gdb)
14              sub x5, x6, x7          # x5 = x6 - x7
=> 0x80000008 <_start+8>:       b3 02 73 40     sub     t0,t1,t2
1: /z $x5 = 0x80000000
2: /z $x6 = 0xffffffff
3: /z $x7 = 0xfffffffe
(gdb)
stop () at test.s:17
17              j stop                  # Infinite loop to stop execution
=> 0x8000000c <stop+0>: 6f 00 00 00     j       0x8000000c <stop>
1: /z $x5 = 0x00000001
2: /z $x6 = 0xffffffff
3: /z $x7 = 0xfffffffe
(gdb)
```


Demontáž pomocí kódu značky :

```
test.elf:     file format elf32-littleriscv


Disassembly of section .text:

80000000 <_start>:

        .text                   # Define beginning of text section
        .global _start          # Define entry _start

_start:
        li x6, -1               # x6 = -1
80000000:       fff00313                li      t1,-1
        li x7, -2               # x7 = -2
80000004:       ffe00393                li      t2,-2
        sub x5, x6, x7          # x5 = x6 - x7
80000008:       407302b3                sub     t0,t1,t2

8000000c <stop>:

stop:
        j stop                  # Infinite loop to stop execution
:
```


Zkuste rozebrat b3 05 95 00 (nízká adresa na vysokou adresu):

Skutečný příkaz: 00 95 05 b3 = "0000 0000 1001 0101 0000 0101 1011 0011




```
0000 000/0 1001 /0101 0/000/ 0101 1/011 0011
funct7/rs2/rs1/funct3/rd/opcode
```

Z porovnávací tabulky je vidět, že registry pro operace sčítání jsou x9, x10, x11 a x11=x9+x10.

2. Nová instrukce ADDI

imm[11:0]		rs1	110	rd	0010011	I ori
imm[11:0]		rs1	111	rd	0010011	I andi
0000000	shamt	rs1	001	rd	0010011	I slli

Ve skutečnosti chci představit příkaz Itype. Rtype provozuje tři registry, zatímco Itype má pouze dva registry a má 12bitovou okamžitou hodnotu (ekvivalent nepřítomnosti funct7 a rs2).

To může **snížit činnost registru** , to znamená, že **data lze přímo použít v instrukci** , ale protože existuje pouze 12 bitů, **rozsah výrazu je velmi omezený** , s výjimkou bitu znaménka, to znamená v [-2048 ,2047] uvnitř. Při účasti na výpočtu bude také **znaménko rozšířeno** na 32 bitů (zde pro RV32I). Stojí za zmínku, že **RISCV ISA neposkytuje instrukci SUBI** .

Další pseudoinstrukce implementované na základě instrukcí aritmetických operací

NEG: záporná, inverzní operace, NEG RD RS je ekvivalentní SUB RD X0 RS (0-RS je vložen do RD.)

MV: pohyb, pohyb, PŘIDEJ RD 0 RS (přidej 0 k RS a dej to do RD.)

NOP: Nedělejte nic, to znamená, že pro ADDI x0 x0 0 vložte 0+x0 do x0. x0 je pouze pro čtení a obsahuje pouze 0s.

Potom je rozsah přiřazení hodnoty adi omezený a 12bitová data nemohou splnit naše požadavky na přiřazení. U 32bitových registrů můžeme upravit instrukci tak, aby nejprve přiřadila horních 20 bitů a poté pomocí addi přiřadila spodních 12 bitů.

LUI instrukce (okamžité načtení horní části)

Ve skutečnosti je to Utype zodpovědný za poskytování horních 20 bitů rs1 (zde rd).

operační kód: 0110111

31 25	24 20	19 15	14 12	11 7	6 0	
imm[31:12]				rd	0110111	U lui
imm[31:12]				rd	0010111	U auipc

Příklad: lui x5 0x12345, poté je horních 20 bitů x5 0001 0010 0011 0100 0101 a spodních 12 bitů je 0.

Pojďme se tedy podívat, jak pomocí této instrukce spolupracovat s addi k dokončení načítání velkých čísel v registrech.

Můžete si myslet, že jeden je odpovědný za vysokou pozici a druhý za nízkou pozici. Ve skutečnosti tomu tak není. Uveďte dva příklady:

Pokud chcete načíst 12345678, příkaz lui načte 0x12345 addi plus 0x678.

```
12345 678
lui x5 0x12345
0001 0010 0011 0100 0101
addi x5 0x678 x5
0110 0111 1000
0000 0000 0000 0000 0000 0110 0111 1000

最终得到
0001 0010 0011 0100 0101 0110 0111 1000
```

Zdá se, že to jde dobře, že? Podívejme se na další příklad:

```
12345 fff
lui x5 0x12345
0001 0010 0011 0100 0101 0000 0000 0000
addi x5 0xfff x5
0110 0111 1000
1111 1111 1111 1111 1111 1111 1111 1111

最终得到

 (0001)  0001 0010 0011 0100 0100 1111 1111 1111
溢出不算：得到什么?
原因是符号拓展。
```

Nakonec je to (1) 12344FFF. Je vidět, že je to úplně jiné, než potřebujeme. Je to proto, že FFF zde přidaný addi je ve skutečnosti záporné číslo. Po rozšíření znaménka je to úplně jiné, než jsme si představovali To znamená, že je podobný tomuto principu: V systému, kde maximální hodnota je 100 a přeteče na 0, se odečtení 1 od čísla rovná sečtení 99, například 2-1=1, a přičtení 99 je ekvivalent k přičtení 100 a následnému odečtení 1. Pro systém Řeknutí 100 je ekvivalentní přetečení 0 znamená, že výsledek 101 je stejný jako 01.

Takže pokud chceme získat číslo 12345fff, nemůžeme jednoduše přidat x5 0xfff x5 přímo a fff, které chceme, ve skutečnosti představuje část kladné hodnoty. Pak, podle obráceného myšlení právě teď,

když přidáme fff , rozšíří se o znaménko Poté, jako jiný druh odčítání (přidání fff způsobí, že se předchozí znaménko rozšíří o 1....1, aby přední přeteklo.), potřebujeme pouze přidat 1 k vysoké nejprve bit a pak odečteme vzdálenost od čísla, které chceme právě teď přidat Rozdíl přetečení, to znamená, pokud chceme přidat fff nebo ffe, pak dáme 12346 na pozici high a pak odečteme 1 nebo 2 dosáhnout našeho cíle.

Pojďme to vyřešit:

> 因为加上一个带有符号拓展为
> *1111 1111 1111 1111 1111* xxxx xxxx xxxx 数
> 虽然最后低位被补全，但是高位发生了溢出
> （相当于一个最大两位数的系统，比如8加上99就会变为107，得到07，
> 相当于做了一个减法，减去的是99和最大溢出上限的差距 1 。）
>
> 所以我们要构造一个具有
> *xxxx x*xxx xxxx xxxx xxxxx 1xxx xxxx xxxx的数字，
> 就会出现问题，那么我们怎么办呢，同样的我们也利用了所谓溢出的原理，
> 既然加99使其溢出相当于减1.那么我们提前在高位加上1，

Ve skutečnosti je to trochu komplikované a někdy je to těžké pochopit, takže máme pseudoinstrukci: li (okamžité načtení). Assembler vygeneruje správnou skutečnou instrukci podle skutečné situace.

```
# Load Immediate
# Format:
#   LI RD, IMM
# Description:
#   The immediate value (which can be any 32-bit value) is copied into RD.
#   LI is a pseudoinstruction, and is assembled differently depending on
#   the actual value present.
#
#   If the immediate value is in the range of -2,048 .. +2,047, then it can
#   be assembled identically to:
#
#   ADDI RD, x0, IMM
#
#   If the immediate value is not within the range of -2,048 .. +2,047 but
#   is within the range of a 32-bit number (i.e., -2,147,483,648 .. +2,147,483,6
#   then it can be assembled using this two-instruction sequence:
```

```
#
    #   LUI RD, Upper-20
#   ADDI RD, RD, Lower-12
#
#   where "Upper-20" represents the uppermost 20 bits of the value
#   and "Lower-12" represents the least significant 12-bits of the value.
#   Note that, due to the immediate operand to the addi has its
#   most-significant-bit set to 1 then it will have the effect of
#   subtracting 1 from the operand in the lui instruction.

    .text           # Define beginning of text section
    .global _start      # Define entry _start

_start:
    # imm is in the range of [-2,048, +2,047]
    li x5, 0x80

    addi x5, x0, 0x80

    # imm is NOT in the range of [-2,048, +2,047]
    # and the most-significant-bit of "lower-12" is 0
    li x6, 0x12345001

    lui x6, 0x12345
    addi x6, x6, 0x001

    # imm is NOT in the range of [-2,048, +2,047]
    # and the most-significant-bit of "lower-12" is 1
    li x7, 0x12345FFF

    lui x7, 0x12346
    addi x7, x7, -1

stop:
    j stop          # Infinite loop to stop execution

    .end            # End of file
```

To je smysl směrnice.

li rd, immediate x[rd] = immediate

立即数加载 *(Load Immediate)*. 伪指令(Pseudoinstruction), RV32I and RV64I.

使用尽可能少的指令将常量加载到 x[*rd*]中。在 RV32I 中，它等同于执行 **lui** 和/或 **addi**；对于 RV64I，会扩展为这种指令序列 **lui, addi, slli, addi, slli, addi ,slli, addi**。


```
        .text                   # Define beginning of text section
        .global _start          # Define entry _start

_start:
        # imm is in the range of [-2,048, +2,047]
        li x5, 0x80
80000000:       08000293                li      t0,128

        addi x5, x0, 0x80
80000004:       08000293                li      t0,128

        # imm is NOT in the range of [-2,048, +2,047]
        # and the most-significant-bit of "lower-12" is 0
        li x6, 0x12345001
80000008:       12345337                lui     t1,0x12345
8000000c:       00130313                addi    t1,t1,1 # 12345001 <_start-0x6dc
bafff>

        lui x6, 0x12345
80000010:       12345337                lui     t1,0x12345
        addi x6, x6, 0x001
80000014:       00130313                addi    t1,t1,1 # 12345001 <_start-0x6dc
bafff>

        # imm is NOT in the range of [-2,048, +2,047]
        # and the most-significant-bit of "lower-12" is 1
        li x7, 0x12345FFF
80000018:       123463b7                lui     t2,0x12346
8000001c:       fff38393                addi    t2,t2,-1 # 12345fff <_start-0x6d
cba001>

        lui x7, 0x12346
80000020:       123463b7                lui     t2,0x12346
        addi x7, x7, -1
80000024:       fff38393                addi    t2,t2,-1 # 12345fff <_start-0x6d
cba001>

80000028 <stop>:
:
```


Porovnáním generovaných strojových kódů můžeme zjistit, že li generuje různé doraemony při zpracování různých čísel, přesně jak jsme očekávali. Je vidět, že generované li je stejné, jak jsme očekávali, což má za následek, že se stejný strojový kód objeví dvakrát.

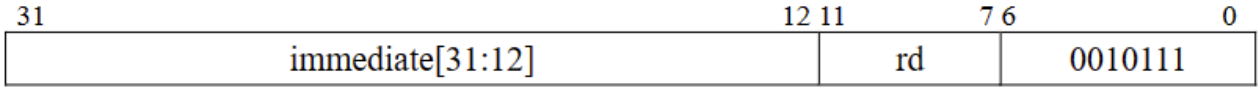
```
00000000  93 02 00 08 93 02 00 08  37 53 34 12 13 03 13 00  |........7S4.....|
00000010  37 53 34 12 13 03 13 00  b7 63 34 12 93 83 f3 ff  |7S4......c4.....|
00000020  b7 63 34 12 93 83 f3 ff  6f 00 00 00              |.c4.....o...|
0000002c
```


3. Stavební adresa AUIPC

Hovořili jsme o sčítání a odčítání aritmetiky a sestavování čísel. Pak je adresa také 32 bitů, co máme dělat, když potřebujeme také zkonstruovat adresu v registru? auipc je:

auipc rd, immediate x[rd] = pc + sext(immediate[31:12] << 12)

PC 加立即数 (Add Upper Immediate to PC). U-type, RV32I and RV64I.
把符号位扩展的 20 位（左移 12 位）立即数加到 *pc* 上，结果写入 x[*rd*]。



Myslel jsem, že je to skutečný příkaz, ale chatgpt to pro mě přeložil na pseudopříkaz. Jen se tomu zasměj. Stručně řečeno, funkcí auipc je přidat okamžitý výsledek čtení dat k vysokým 20 bitům pc a uložit je do určeného registru. Zde, pokud je okamžitá hodnota 0, lze přímo číst hodnotu pc.

Existují také pseudo-instrukce, la, což je příkaz načíst adresu, stejně jako li:

la rd, symbol x[rd] = &symbol

地址加载 *(Load Address)*. 伪指令(Pseudoinstruction), RV32I and RV64I.
将 *symbol* 的地址加载到 x[*rd*]中。当编译位置无关的代码时，它会被扩展为对全局偏移量表(Global Offset Table)的加载。对于 RV32I，等同于执行 **auipc** rd, offsetHi，然后是 **lw** rd, offsetLo(rd)；对于 RV64I，则等同于 **auipc** rd, offsetHi 和 **ld** rd, offsetLo(rd)。如果 offset 过大，开始的算加载地址的指令会变成两条，先是 **auipc** rd, offsetHi 然后是 **addi** rd, rd, offsetLo。

Ve skutečnosti jde o přiřazení adresy štítku rd.

```
la x5, _start
jr x5
```

jr znamená skočit na adresu uloženou v x5.

4. Logické provozní pokyny

a,nebo,xor.andi.ori,xori.

A, NEBO, VÝHRADNÍ NEBO, OKAMŽITĚ A, OKAMŽITÉ NEBO, OKAMŽITĚ VÝLUČNÉ NEBO.

```
and rd, rs1,rs2  #R类

andi rd,rs1,imm  #I类
```

Ostatní jsou na tom podobně.

Navíc zde není žádná negační instrukce, pouze pseudoinstrukce, využívající operace XOR

not command: ekvivalentní xor rd, rs1, -1. -1 je celá 1 a XOR způsobí, že všechna 1 se stanou 0 a všechny 0 až 1, aby se dosáhlo inverzního efektu. Pokud jde o to, proč ne všechna 0, všechna nula znamená zachovat původní tvar. Exclusive or znamená, že je 0 stejně jako já, pak 0 je vždy 0 a rozdíl oproti nebo je v tom, že dvě jedničky se nemohou objevit současně.

Pět, instrukce pro obsluhu směny

To je bolest hlavy pro mnoho studentů, kteří studovali počítací skupinu. Pojďme si nejprve promluvit o jednoduché.

Logický chod směny:

sll, srl, slli, srli.

```
sll rd, rs1,rs2  #R类 rd=rs1<<rs2

slli rd,rs1,imm  #I类 rd=rs1<<imm

srl rd, rs1,rs2  #R类 rd=rs1>>rs2

srli rd,rs1,imm  #I类 rd=rs1>>imm
```

Jak logický posun doleva, tak doplněk logického posunu doprava jsou 0.

Další kategorií je aritmetický posun a aritmetický posun lze posunout pouze doprava, což nemusí být stejné jako početní skupina. Dokončení je znaménkový bit, to znamená, že 00100000 bude dokončeno na 00010000 a 10010000 bude dokončeno na 11001000.

Například -2: původní kód je 1000 0010 doplňkový kód: 11111110, po pravém posunu: 11111111;-1.

V 64bitové instrukční sadě jsou další posunové instrukce, které zde nebudou uvedeny.

6. Instrukce pro čtení a zápis do paměti

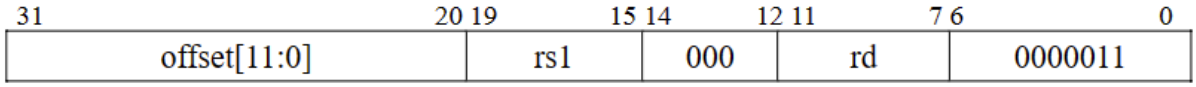
Chcete-li zobrazit příručku, podívejte se do návodu:

I znamená zatížení bhwd je 1 byte, 2 byty, 4 byty, 8 bytů, což znamená byte, polovina, slovo, dvojnásobek. Význam u je expandovat jako číslo bez znaménka Proč lw nenapsalo lwu?Ve skutečnosti existuje, ale není to pro 32bitové RV32I, ale je to pro RV64I. Proč? Můžete o tom přemýšlet.

lb rd, offset(rs1) x[rd] = sext(M[x[rs1] + sext(offset)][7:0])

字节加载 *(Load Byte)*. I-type, RV32I and RV64I.

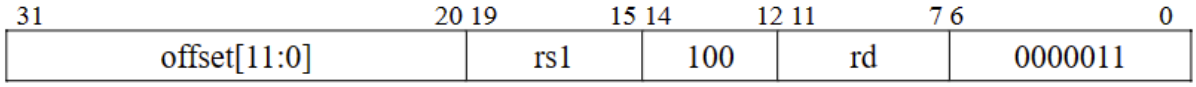
从地址 x[*rs1*] + *sign-extend*(*offset*)读取一个字节，经符号位扩展后写入 x[*rd*]。



lbu rd, offset(rs1) x[rd] = M[x[rs1] + sext(offset)][7:0]

无符号字节加载 *(Load Byte, Unsigned)*. I-type, RV32I and RV64I.

从地址 x[*rs1*] + *sign-extend*(*offset*)读取一个字节，经零扩展后写入 x[*rd*]。

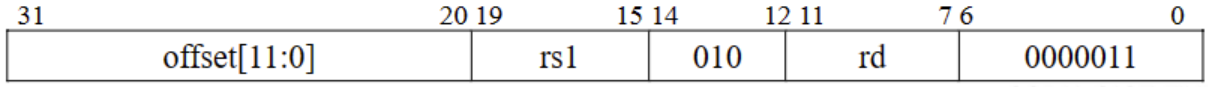


lw rd, offset(rs1) x[rd] = sext(M[x[rs1] + sext(offset)][31:0])

字加载 *(Load Word)*. I-type, RV32I and RV64I.

从地址 x[*rs1*] + *sign-extend*(*offset*)读取四个字节，写入 x[*rd*]。对于 RV64I，结果要进行符号位扩展。

压缩形式：**c.lwsp** rd, offset; **c.lw** rd, offset(rs1)

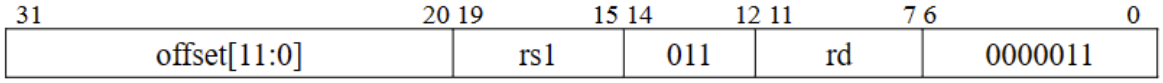


ld rd, offset(rs1) x[rd] = M[x[rs1] + sext(offset)][63:0]

双字加载 *(Load Doubleword)*. I-type, RV32I and RV64I.

从地址 x[*rs1*] + *sign-extend*(*offset*)读取八个字节，写入 x[*rd*]。

压缩形式：**c.ldsp** rd, offset; **c.ld** rd, offset(rs1)

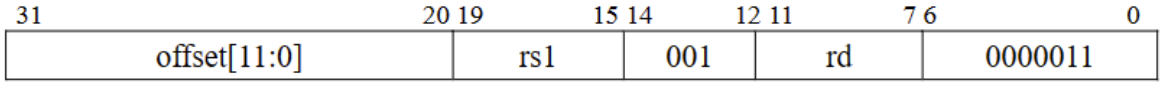


---

lh rd, offset(rs1) x[rd] = sext(M[x[rs1] + sext(offset)][15:0])

半字加载 *(Load Halfword)*. I-type, RV32I and RV64I.

从地址 x[*rs1*] + *sign-extend*(*offset*)读取两个字节，经符号位扩展后写入 x[*rd*]。

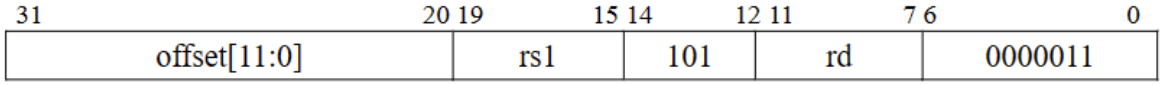


---

lhu rd, offset(rs1) x[rd] = M[x[rs1] + sext(offset)][15:0]

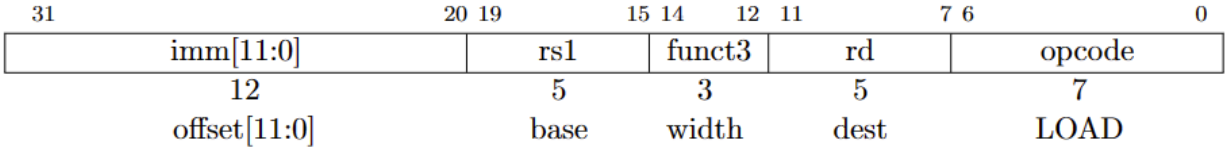
无符号半字加载 *(Load Halfword, Unsigned)*. I-type, RV32I and RV64I.

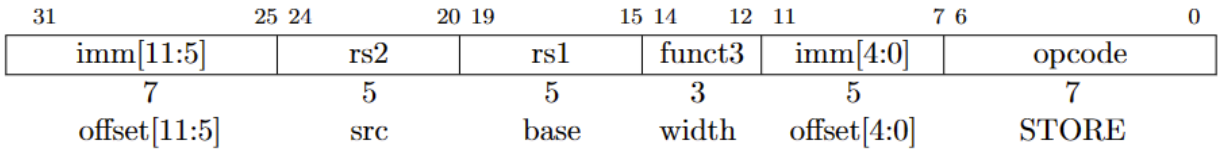
从地址 x[*rs1*] + *sign-extend*(*offset*)读取两个字节，经零扩展后写入 x[*rd*]。



---

less-significant register bytes.





Vezměte prosím na vědomí, že jsem neuvedl všechny příkazy, musíte spíše porozumět významu názvu a typu příkazu, než doufat, že zde najdete manuál.

Souhrnná funkce spočívá v načítání čísel z externí paměti do registrů.

lb rd,imm(rs1)

Tento příkaz má najít paměť podle adresy dané rs1 plus offset daný imm a pak načíst bajt dat do rd s **příponou znaménka .** Ostatním příkazům rozumíte sami, takže nebudu zacházet do podrobností. A protože jsou čteny bity imm12, existuje maximálně offset [-2048, 2047] vzhledem k adrese uložené v rs.

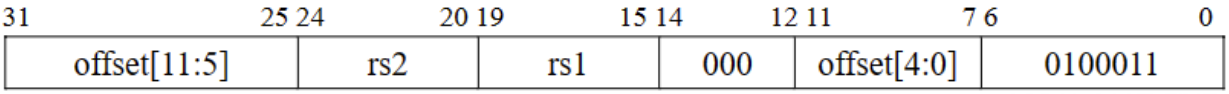
Paměť musíme nejen číst, ale i zapisovat, podobně, ale nemusíme myslet na symboly, protože při ukládání do paměti není potřeba rozšiřování.

SB, SH, SW

sb rs2, offset(rs1) $M[x[rs1] + sext(offset) = x[rs2][7:0]$

存字节(Store Byte). S-type, RV32I and RV64I.

将 x[*rs2*]的低位字节存入内存地址 x[*rs1*]+*sign-extend(offset)*。

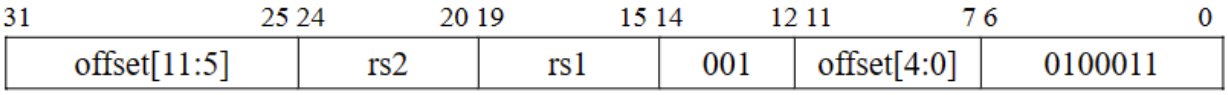


31 25	24 20	19 15	14 12	11 7	6 0
offset[11:5]	rs2	rs1	000	offset[4:0]	0100011

sh rs2, offset(rs1) $M[x[rs1] + sext(offset) = x[rs2][15:0]$

存半字(Store Halfword). S-type, RV32I and RV64I.

将 x[*rs2*]的低位 2 个字节存入内存地址 x[*rs1*]+*sign-extend(offset)*。



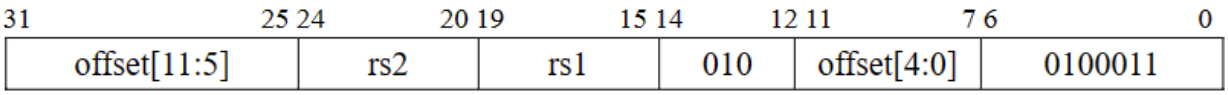
31 25	24 20	19 15	14 12	11 7	6 0
offset[11:5]	rs2	rs1	001	offset[4:0]	0100011

sw rs2, offset(rs1) $M[x[rs1] + sext(offset) = x[rs2][31:0]$

存字(Store Word). S-type, RV32I and RV64I.

将 x[*rs2*]的低位 4 个字节存入内存地址 x[*rs1*]+*sign-extend(offset)*。

压缩形式：**c.swsp** rs2, offset; **c.sw** rs2, offset(rs1)



31 25	24 20	19 15	14 12	11 7	6 0
offset[11:5]	rs2	rs1	010	offset[4:0]	0100011

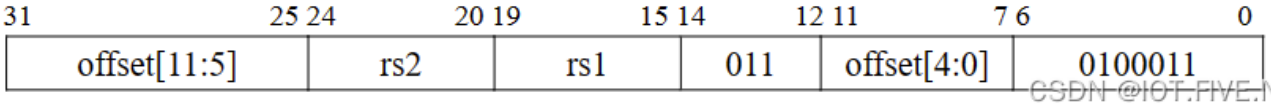
64 bit také:

sd rs2, offset(rs1) $M[x[rs1] + sext(offset) = x[rs2][63:0]$

存双字(Store Doubleword). S-type, RV64I only.

将 x[*rs2*]中的 8 字节存入内存地址 x[*rs1*]+*sign-extend(offset)*。

压缩形式： **c.sdsp** rs2, offset; **c.sd** rs2, offset(rs1)



například:

```
sb rs2,imm(rs1)
```

Vezměte spodních 8 bitů z rs2 a uložte je do paměti na adresu rs1+imm. Všimněte si, že pokud je paměť zarovnána po bytech, bude to fungovat.

Kompilace bude dokončena v další části.