# RISC-V

Robert Baruch
robert.c.baruch@gmail.com

November 8, 2018

# Chapter 1

# Harts and the FENCE instruction

A hart is a hardware thread. This is in contrast to a software thread. A hart is an independent core with its own set of registers and its own program counter. However, all cores share the entire memory space.

Each hart executes its own instructions one by one, in order. However, because of caches and other accelerator magic, some other hart may not see the loads and stores from this hart in the same order! For example, imagine that hart 1 executes instructions in this order:

Store 0 to location A Store 1 to location B Read location A Read location B Add them. Store the result to location C.

In any sane universe, 2 should be written to location C, and as far as hart 1 is concerned, that is exactly what happens.

Next, imagine that some hart is monitoring location C by reading it over and over. When it sees that location C changes, it immediate reads A and B, adds them, and compares the result with C. Maybe hart 2 is testing that hart 1 is actually adding A and B properly.

However, now imagine that hart 1 is actually accessing a fast memory cache. All the instructions are reading and writing that cache. Eventually, the cache gets flushed to main memory. However, the order that the cache writes to memory is not specified! It could write location C first, then B, then A. The cache doesn't care about the order of operations, it just cares about the view of memory according to hart 1 and the changes it made. So hart 2 might actually see 2 in C but 5 in B and 42 in A because the cache hasn't gotten around to writing locations A and B yet. The addition doesn't compare, and hart 2 thinks hart 1 is malfunctioning.

This is the essence of the *relaxed memory model*. If two harts access different areas of memory, no problem exists. But when they access the same areas of memory, things can go a bit wonky.

The FENCE instruction forces an ordering on the cache. For example, a FENCE instruction might dictate that previous writes must occur before subsequent writes. This means that a cache would be forced to write everything before the FENCE before writ-

ing anything after the FENCE. Thus, the cache could only end up writing in either of these two orders: A, B, C, or B, A, C. That is, the FENCE instruction declares that every other hart is guaranteed to see C written only after A and B are written.

So with such a FENCE, hart 2 will not see inconsistent values.

The `FENCE.I` instruction has to do with instruction caches and pipelines. If a hart were to write to memory which is actually code, it is possible that a cache or pipeline could still contain the old value at that address, meaning that the instruction to execute at that address is incorrect. Executing a `FENCE.I` instruction guarantees that the hart's caches and pipelines are updated with respect to instructions to execute.

# Chapter 2

# The Instructions

## 2.1 Abbreviations

| | |
|---|---|
| XLEN | The processor's integer register width |
| imm | The unsigned immediate value in the instruction |
| int12 | 12-bit signed integer |
| uint12 | 12-bit unsigned integer |
| int13e | 13-bit even signed integer |
| uint20 | 20-bit unsigned integer |
| int21e | 21-bit even signed integer |
| uintN | $\log_2$(XLEN)-bit unsigned integer |
| pc | The program counter |
| rd | The destination register |
| rs1 | Source register 1 |
| rs2 | Source register 2 |
| sext($x$) | $x$, sign-extended to XLEN bits |
| zext($x$) | $x$, zero-extended to XLEN bits |
| (r) | The contents of a register, as a signed XLEN-bit integer |
| $(r)_u$ | The contents of a register, as an unsigned XLEN-bit integer |
| $X_b$ | The lowest 8 bits of $X$ |
| $X_h$ | The lowest 16 bits of $X$ |
| $X_w$ | The lowest 32 bits of $X$ |
| $X_d$ | The lowest 64 bits of $X$ |
| $X_n$ | The lowest $\log_2$(XLEN) bits of $X$ |
| $[a]_b$ | The 8-bit integer stored in memory at address $a$ |
| $[a]_h$ | The 16-bit integer stored in memory at address $a$ |
| $[a]_w$ | The 32-bit integer stored in memory at address $a$ |
| $[a]_d$ | The 64-bit integer stored in memory at address $a$ |
| $\leftarrow$ | Receives the value of |
| $\overline{1}$ | All ones, except the least significant bit is zero |
| $+$ | Addition |
| $-$ | Subtraction |
| $<<$ | Binary shift left |

| | |
|---|---|
| $>>$ | Binary signed shift right |
| $>>_u$ | Binary unsigned shift right |
| $\wedge$ | Logical bitwise and |
| $\vee$ | Logical bitwise or |
| $\oplus$ | Logical bitwise exclusive or (xor) |

# ADD

Add (register/register)

---

**Instruction Format:** R

**Opcode Type:** OP

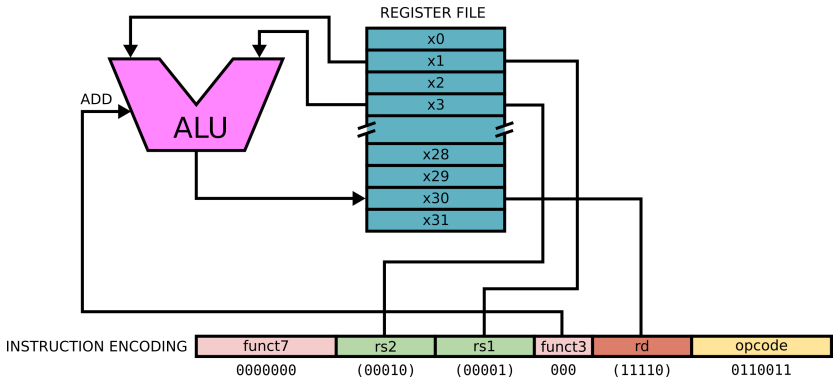
**Assembly:** `ADD rd, rs1, rs2`

**Function:** `(rd) ← (rs1) + (rs2)`

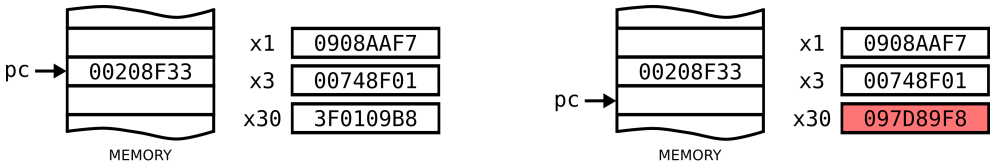
**Description:**

Adds the contents of register `rs1` to the contents of register `rs2`. The result is placed in register `rd`.

**Data Path:**




**Example:** `ADD x30, x1, x3`




BEFORE AFTER

**Notes:**

Overflows of the `ADD` instruction for signed arithmetic can be checked using `SLTI` and `SLT`:

```
ADD   x1, x2, x3
SLTI  x4, x3, 0
SLT   x5, x1, x2
BNE   x4, x5, overflow
```

# ADDI

Add (register/immediate)

---

**Instruction Format:** I

**Opcode Type:** OP-IMM

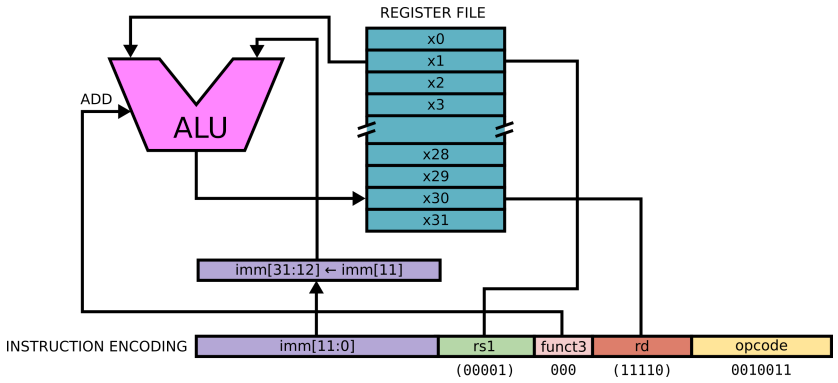
**Assembly:** `ADDI rd, rs1, int12`

**Function:** `(rd) ← (rs1) + sext(imm)`

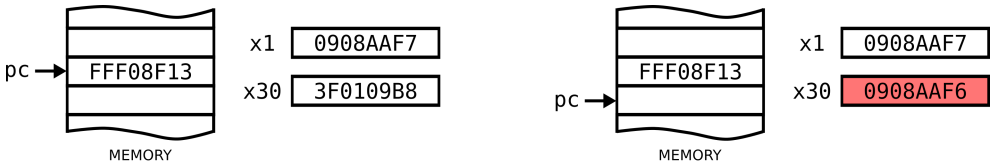
**Description:**

Adds the sign-extended immediate value to the contents of register `rs1`. The result is placed in register `rd`.

**Data Path:**




**Example:** `ADDI x30, x1, -1`




BEFORE AFTER

**Notes:**

There is no separate `SUBI` instruction. However, `SUBI` is a psuedo-instruction encoded as `ADDI` where the immediate value is negated. Thus, the following two instructions are encoded the same way:

```
SUBI  x30, x1, 1
ADDI  x30, x1, -1
```

# AND

Bitwise and (register/register)

---

**Instruction Format:** R

**Opcode Type:** OP

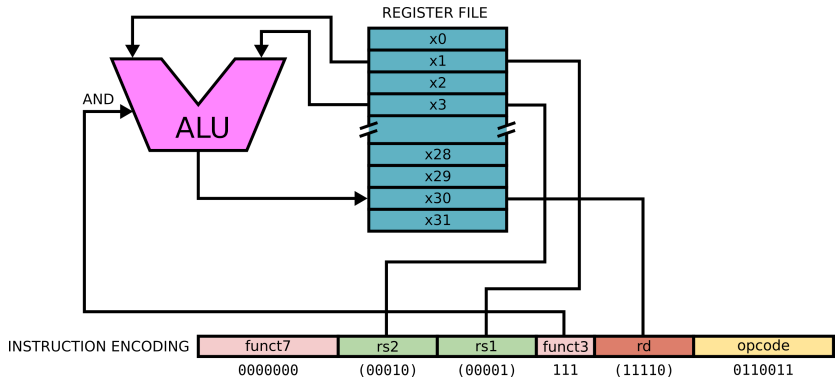
**Assembly:** `AND rd, rs1, rs2`

**Function:** `(rd)` $\leftarrow$ `(rs1)` $\wedge$ `(rs2)`

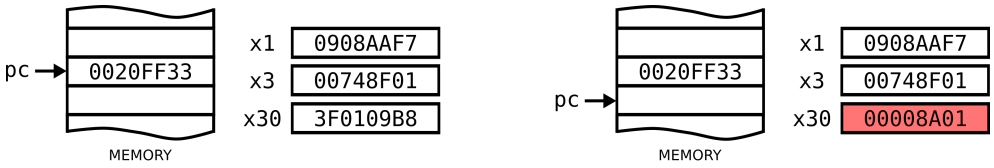
**Description:**

Performs a bitwise AND between the contents of register `rs1` and the contents of register `rs2`. The result is placed in register `rd`.

**Data Path:**




**Example:** `AND x30, x1, x3`



# ANDI

Bitwise and (register/immediate)

---

**Instruction Format:** I

**Opcode Type:** OP-IMM

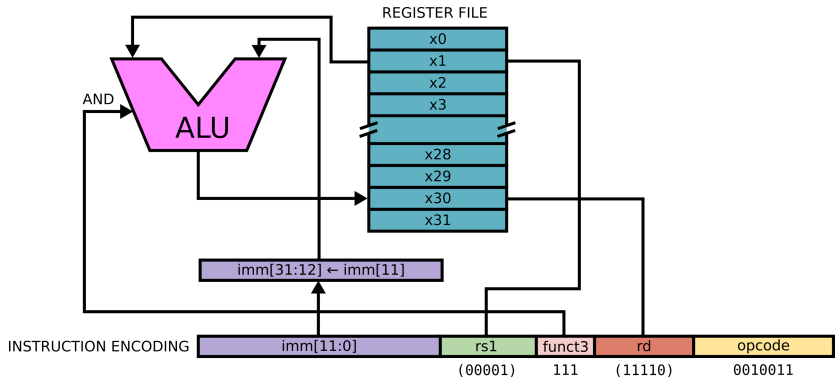
**Assembly:** `ANDI rd, rs1, int12`

**Function:** `(rd) ← (rs1) ∧ sext(imm)`

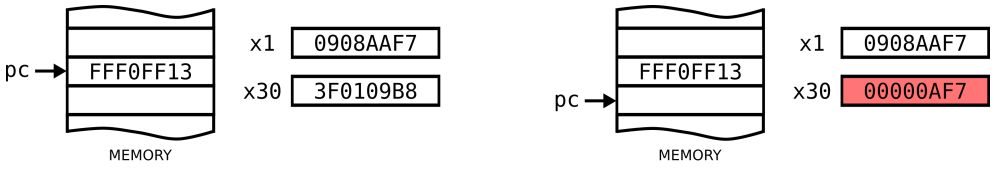
**Description:**

Performs a bitwise AND between the sign-extended immediate value and the contents of register `rs1`. The result is placed in register `rd`.

**Data Path:**




**Example:** `ANDI x30, x1, -1`




BEFORE AFTER

# AUIPC

Add upper immediate to PC

**Instruction Format:** U

**Opcode Type:** AUIPC

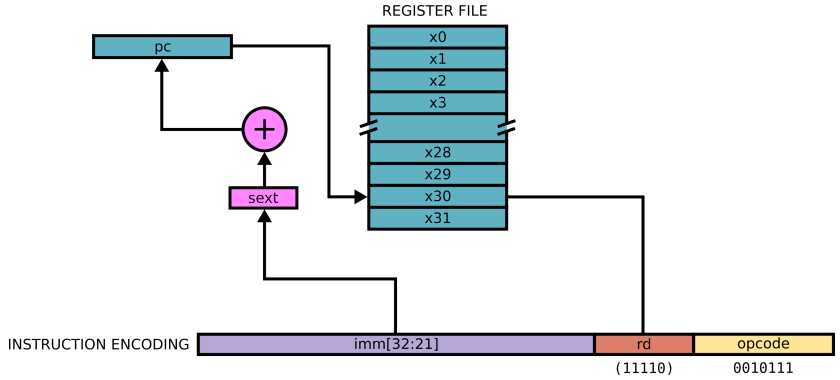
**Assembly:** `AUIPC rd, uint20`

**Function:** `(rd) ← (pc) + sext(imm << 12)`

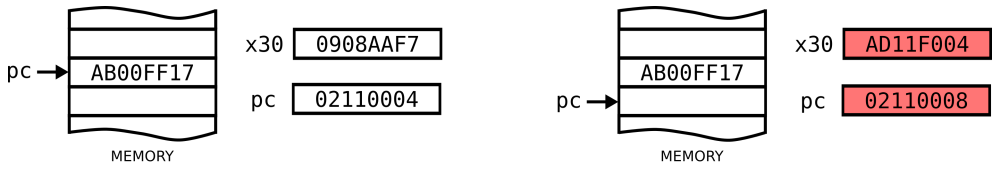
**Description:**

Constructs a 32-bit integer whose upper 20 bits are the immediate value, filling the lower 12 bits with zeros. The result is then sign-extended to XLEN bits, added to the program counter, and the result placed in the destination register.

**Data Path:**




**Example:** `AUIPC x30, 0xAB00F`




BEFORE AFTER

**Notes:**

The AUIPC instruction is used to construct pc-relative offsets outside the 12-bit range.

# BEQ

Branch if equal

**Instruction Format:** B

**Opcode Type:** BRANCH

**Assembly:** `BEQ rs1, rs2, int13e`

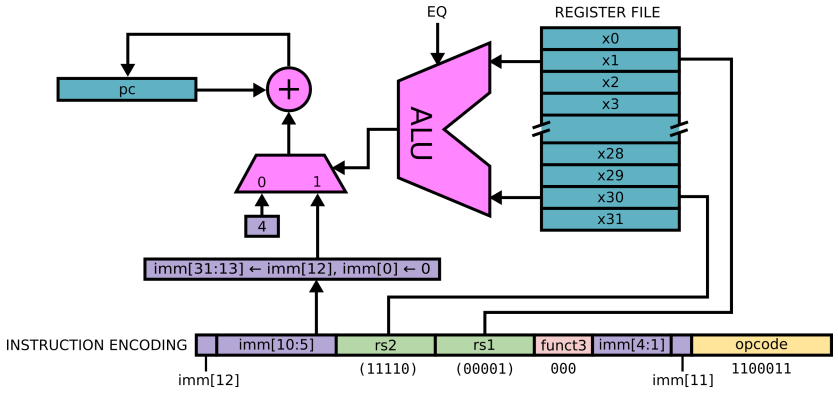
**Function:**

$$(pc) \leftarrow \begin{cases} (pc) + sext(imm) & \text{if } (rs1) = (rs2) \\ (pc) + 4 & \text{otherwise} \end{cases}$$

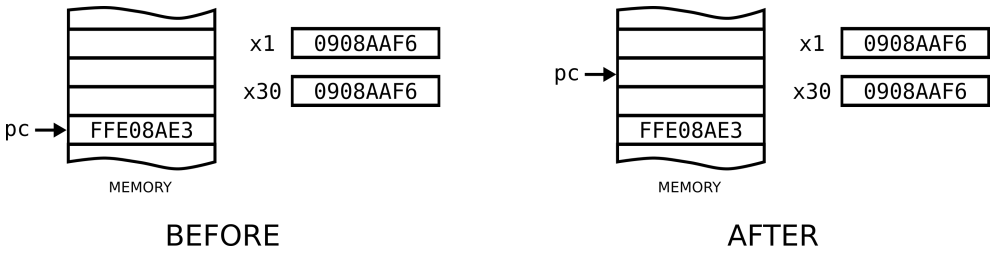
**Description:**

Compares the contents of register `rs1` to the contents of register `rs2`. If they are equal, the signed offset is added to the current program counter, otherwise the program counter is set to the address of the next instruction.

**Data Path:**




**Example:** `BEQ x1, x30, -8`



**Notes:**

Offsets are limited to even addresses in the range [-4096, 4094]. Branches to addresses outside this range can be accomplished using the complementary branch followed by an unconditional jump:

```
BNE  x1, x30, 8
J    far_addr
```

A misaligned instruction fetch exception will be generated if the address of the branch taken is not aligned on a 32-bit boundary, except in processors that support extensions with 16-bit aligned instructions, such as the compressed instruction set extension C.

# BGE

Branch if greater than or equal (signed)

---

**Instruction Format:** B

**Opcode Type:** BRANCH

**Assembly:** `BGE rs1, rs2, int13e`

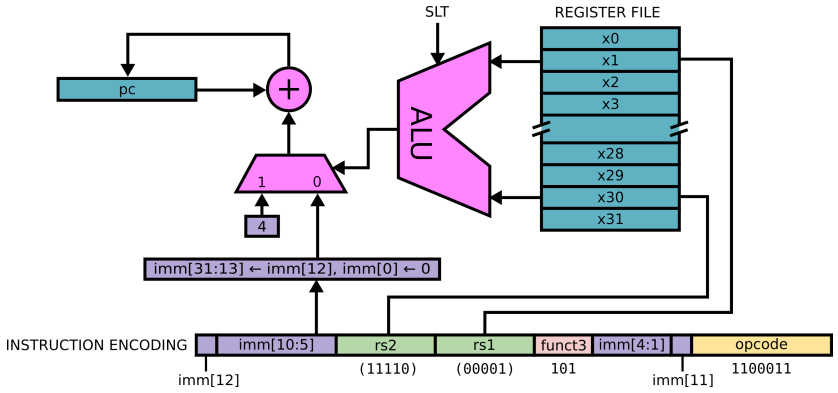
**Function:**

$$(pc) \leftarrow \begin{cases} (pc) + sext(imm) & \text{if } (rs1) \geq (rs2) \\ (pc) + 4 & \text{otherwise} \end{cases}$$

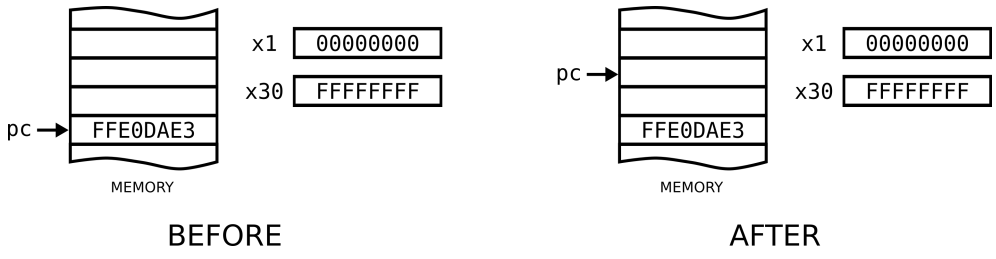
**Description:**

If the signed contents of register `rs1` is greater than or equal to the signed contents of register `rs2`, the signed offset is added to the current program counter, otherwise the program counter is set to the address of the next instruction.

**Data Path:**




**Example:** `BGE x1, x30, -8`



**Notes:**

Offsets are limited to even addresses in the range [-4096, 4094]. Branches to addresses outside this range can be accomplished using the complementary branch followed by an unconditional jump:

```
BLT  x1, x30, 8
J    far_addr
```

A misaligned instruction fetch exception will be generated if the address of the branch taken is not aligned on a 32-bit boundary, except in processors that support extensions with 16-bit aligned instructions, such as the compressed instruction set extension C.

# BGEU

Branch if greater than of equal (unsigned)

---

**Instruction Format:** B

**Opcode Type:** BRANCH

**Assembly:** `BGEU rs1, rs2, int13e`

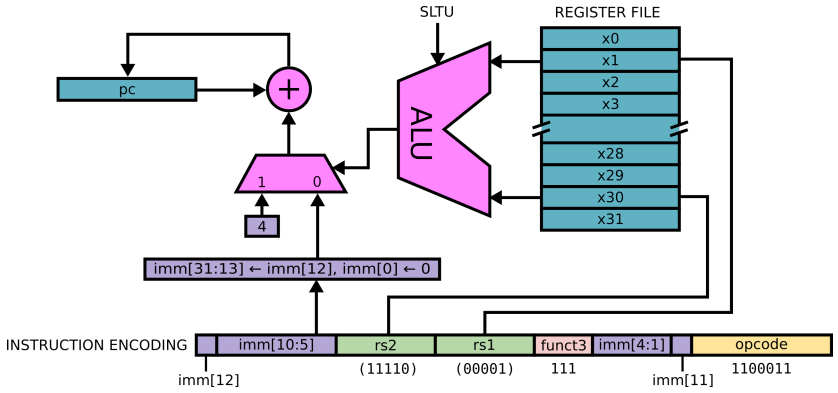
**Function:**

$$(pc) \leftarrow \begin{cases} (pc) + sext(imm) & \text{if } (rs1)_u \geq (rs2)_u \\ (pc) + 4 & \text{otherwise} \end{cases}$$

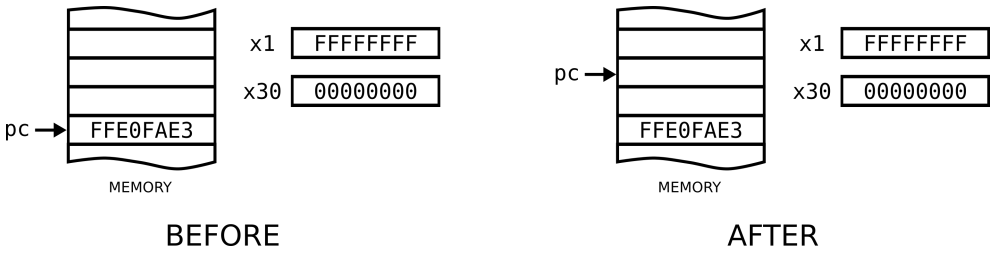
**Description:**

If the unsigned contents of register `rs1` is greater than or equal to the unsigned contents of register `rs2`, the signed offset is added to the current program counter, otherwise the program counter is set to the address of the next instruction.

**Data Path:**




**Example:** `BGEU x1, x30, -8`



**Notes:**

Offsets are limited to even addresses in the range [-4096, 4094]. Branches to addresses outside this range can be accomplished using the complementary branch followed by an unconditional jump:

```
BLTU  x1, x30, 8
J     far_addr
```

A misaligned instruction fetch exception will be generated if the address of the branch taken is not aligned on a 32-bit boundary, except in processors that support extensions with 16-bit aligned instructions, such as the compressed instruction set extension C.

# BLT

Branch if less than (signed)

---

**Instruction Format:** B

**Opcode Type:** BRANCH

**Assembly:** `BLT rs1, rs2, int13e`

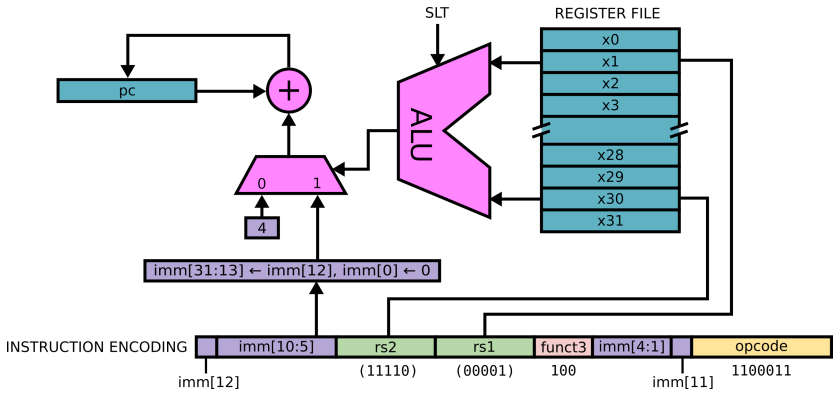
**Function:**

$$(pc) \leftarrow \begin{cases} (pc) + sext(imm) & \text{if } (rs1) < (rs2) \\ (pc) + 4 & \text{otherwise} \end{cases}$$

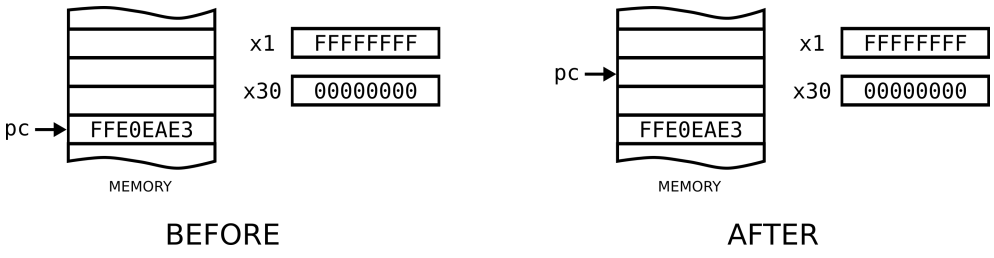
**Description:**

If the signed contents of register `rs1` is less than the signed contents of register `rs2`, the signed offset is added to the current program counter, otherwise the program counter is set to the address of the next instruction.

**Data Path:**




**Example:** `BLT x1, x30, -8`



**Notes:**

Offsets are limited to even addresses in the range [-4096, 4094]. Branches to addresses outside this range can be accomplished using the complementary branch followed by an unconditional jump:

```
BGE  x1, x30, 8
J    far_addr
```

A misaligned instruction fetch exception will be generated if the address of the branch taken is not aligned on a 32-bit boundary, except in processors that support extensions with 16-bit aligned instructions, such as the compressed instruction set extension C.

# BLTU

Branch if less than (unsigned)

---

**Instruction Format:** B

**Opcode Type:** BRANCH

**Assembly:** `BLTU rs1, rs2, int13e`

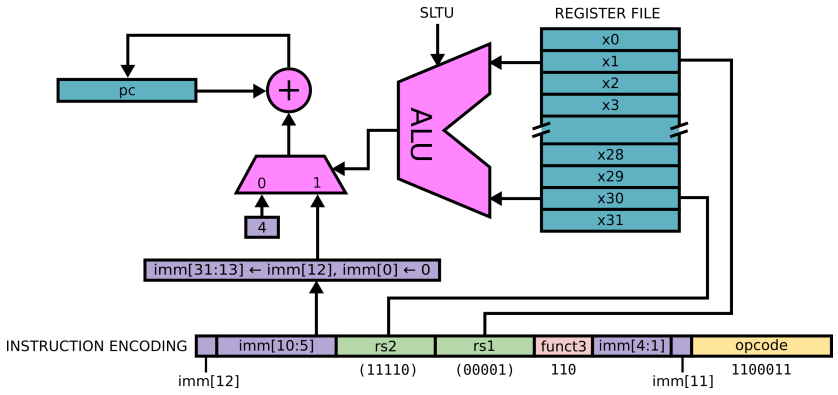
**Function:**

$$(pc) \leftarrow \begin{cases} (pc) + sext(imm) & \text{if } (rs1)_u < (rs2)_u \\ (pc) + 4 & \text{otherwise} \end{cases}$$

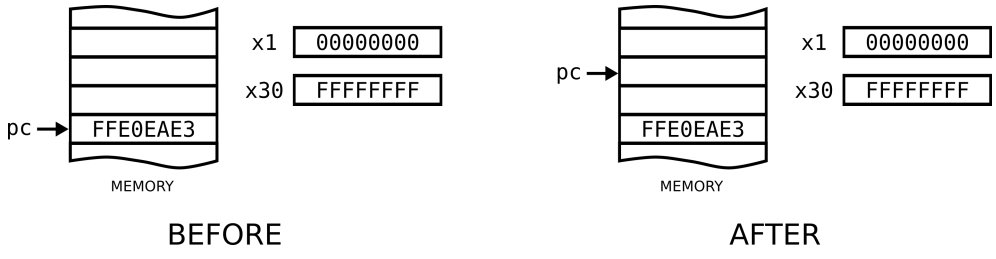
**Description:**

If the unsigned contents of register `rs1` is less than the unsigned contents of register `rs2`, the signed offset is added to the current program counter, otherwise the program counter is set to the address of the next instruction.

**Data Path:**




**Example:** `BLTU x1, x30, -8`



**Notes:**

Offsets are limited to even addresses in the range [-4096, 4094]. Branches to addresses outside this range can be accomplished using the complementary branch followed by an unconditional jump:

```
BGEU  x1, x30, 8
J     far_addr
```

A misaligned instruction fetch exception will be generated if the address of the branch taken is not aligned on a 32-bit boundary, except in processors that support extensions with 16-bit aligned instructions, such as the compressed instruction set extension C.

# BNE

Branch if not equal

**Instruction Format:** B

**Opcode Type:** BRANCH

**Assembly:** `BNE rs1, rs2, int13e`

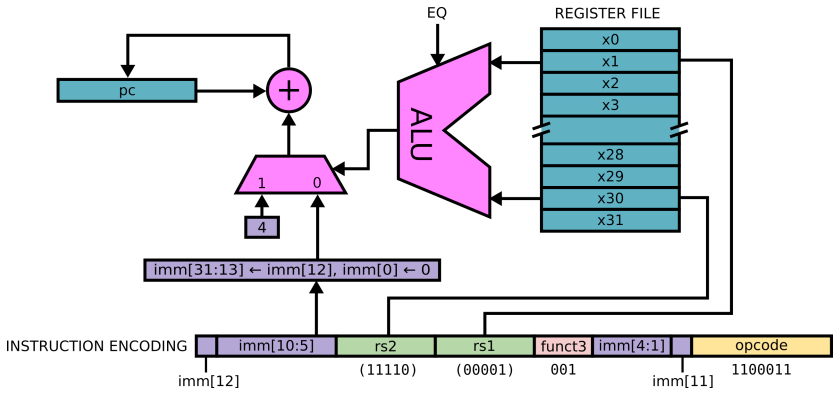
**Function:**

$$(pc) \leftarrow \begin{cases} (pc) + sext(imm) & \text{if } (rs1) \neq (rs2) \\ (pc) + 4 & \text{otherwise} \end{cases}$$

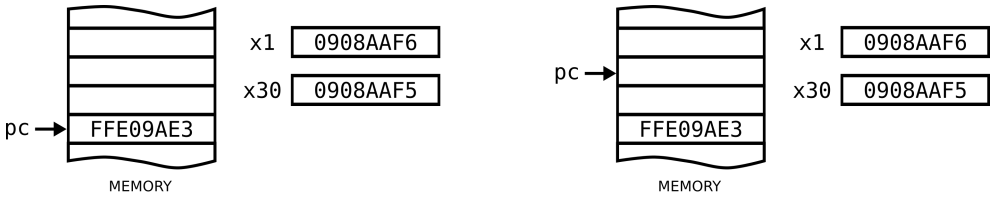
**Description:**

Compares the contents of register `rs1` to the contents of register `rs2`. If they are not equal, the signed offset is added to the current program counter, otherwise the program counter is set to the address of the next instruction.

**Data Path:**




**Example:** `BNE x1, x30, -8`



**Notes:**

Offsets are limited to even addresses in the range [-4096, 4094]. Branches to addresses outside this range can be accomplished using the complementary branch followed by an unconditional jump:

```
BEQ  x1, x30, 8
J    far_addr
```

A misaligned instruction fetch exception will be generated if the address of the branch taken is not aligned on a 32-bit boundary, except in processors that support extensions with 16-bit aligned instructions, such as the compressed instruction set extension C.

# CSRRW

Atomic read/write CSR

**Instruction Format:** I

**Opcode Type:** SYSTEM

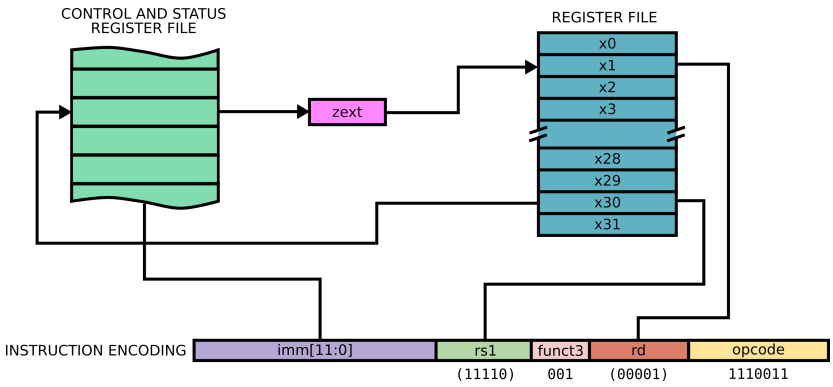
**Assembly:** `CSRRW rd, rs1, csr`

**Function:** `(rd) ← zext(csr[imm]) ← (rs1)`

**Description:**

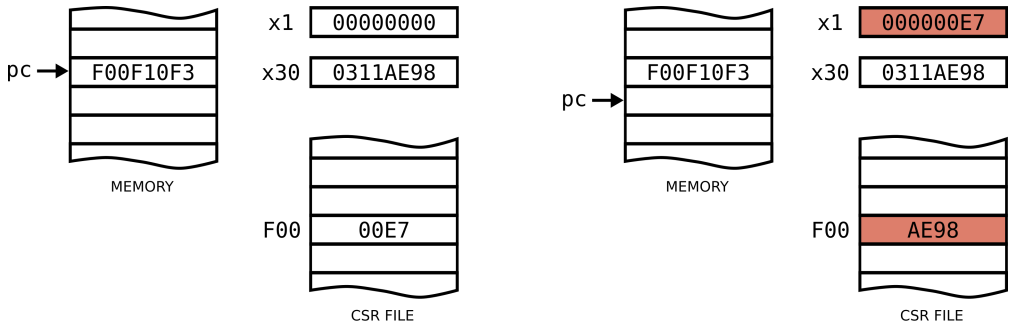
Stores the zero-extended old value of the Control and Status Register in the destination register, and stores the contents of `rs1` in the CSR.

**Data Path:**




**Example:** `CSSRW x1, x30, 0xF00`

In this example, CSR `0xF00` is a 16-bit register.



**Notes:**

If `rd` and `rs1` are the same, this instruction is effectively a swap.

If `rd` is `x0`, then the CSR is not read, and so no side-effect of reading the CSR occurs.

Processor implementations may trap SYSTEM opcodes in a software handler, handle them in hardware, or a mix of both.

# JAL

Jump and link

**Instruction Format:** J

**Opcode Type:** JAL

**Assembly:** `JAL rd, int21e`

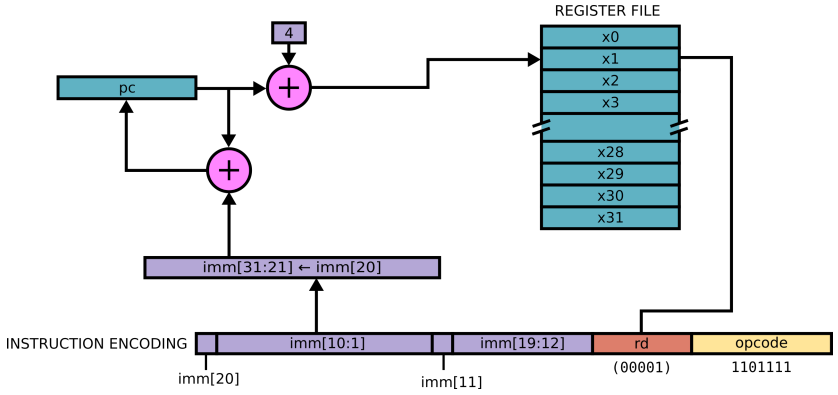
**Function:**

```
(rd) ← (pc) + 4
(pc) ← (pc) + sext(imm)
```

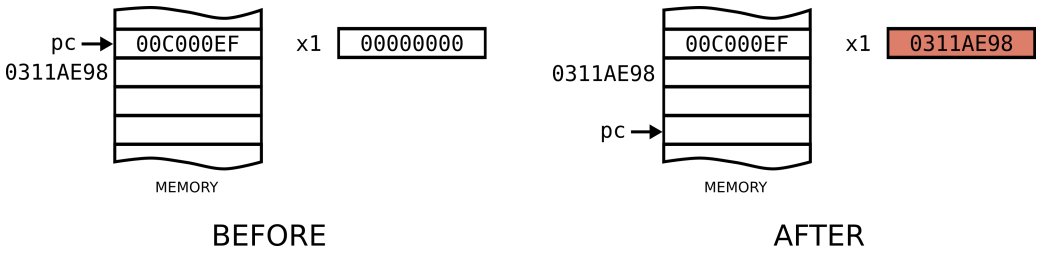
**Description:**

The address of the next instruction is written to `rd`, and the signed offset is added to the current program counter.

**Data Path:**




**Example:** `JAL x1, 12`



**Notes:**

Offsets are limited to even addresses in the range [-1048576, 1048574]. Jumps to relative addresses outside this range can be accomplished using an extra register and the `AUIPC` and `JALR` instructions. For example, jumping to `pc+0x76543210` can be implemented as follows:

```
AUIPC  x30, 0x76543000
JALR   x1, 0x20C(x30)
```

A misaligned instruction fetch exception will be generated if the target address is not aligned on a 32-bit boundary, except in processors that support extensions with 16-bit aligned instructions, such as the compressed instruction set extension C.

# JALR

Jump and link register

**Instruction Format:** I

**Opcode Type:** JALR

**Assembly:** `JALR rd, int12(rs1)`

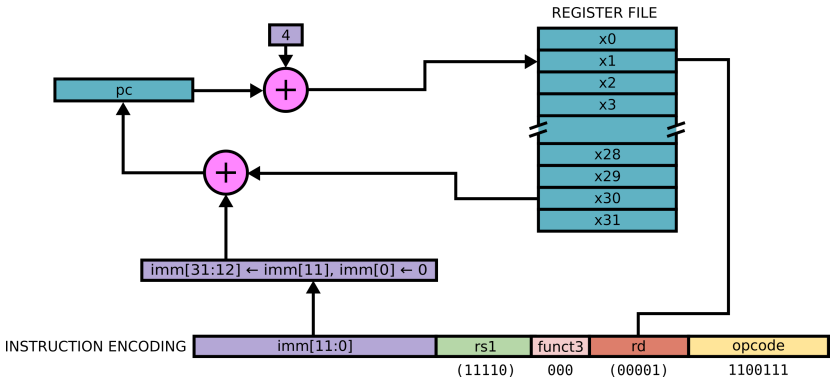
**Function:**

$$(rd) \leftarrow (pc) + 4$$
$$(pc) \leftarrow (pc) + ((rs1) + sext(imm)) \wedge \overline{1}$$

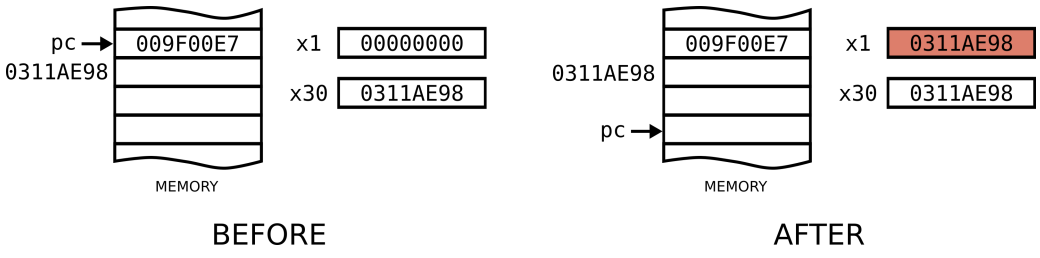
**Description:**

The address of the next instruction is written to `rd`. The signed offset is added to the contents of `rs1`, and the result's least significant bit is set to zero. Then the result is loaded into the program counter.

**Data Path:**




**Example:** `JALR x1, 9(x30)`



**Notes:**

Offsets are limited to even addresses in the range [-2048, 2046]. Jumps to relative indirect addresses outside this range can be accomplished using an extra register. For example, jumping to `0x76543210(x30)` can be implemented as follows:

```
LUI   x29, 0x76543
ADD   x29, x29, x30
JALR  x1, 0x210(x29)
```

A misaligned instruction fetch exception will be generated if the target address is not aligned on a 32-bit boundary, except in processors that support extensions with 16-bit aligned instructions, such as the compressed instruction set extension C.

# LB

Load byte

**Instruction Format:** I

**Opcode Type:** LOAD

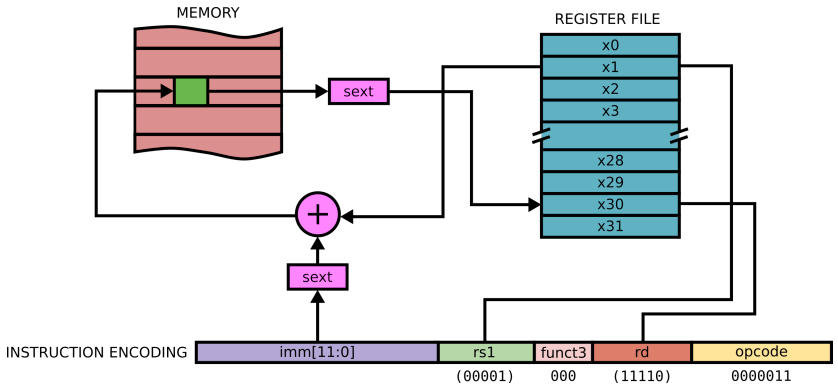
**Assembly:** `LB rd, int12(rs1)`

**Function:** `(rd) ← sext([(rs1) + sext(imm)]_b)`

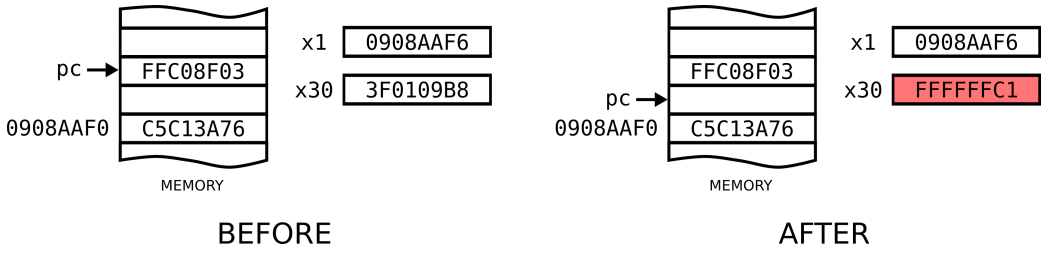
**Description:**

Adds the signed offset to the contents of register `rs1`, treating it as a memory address. Retrieves a byte from memory at that address, and sign-extends it. The result is placed in the destination register.

**Data Path:**




**Example:** `LB x30, -4(x1)`




In this example, the little-endian layout in memory at address `0908AAF0` is `76 3A C1 C5`. The address to access is `0908AAF2`, which contains the byte `C1`. When sign-extended to 32 bits, this is `FFFFFFC1`.

# LBU

Load byte unsigned

**Instruction Format:** I

**Opcode Type:** LOAD

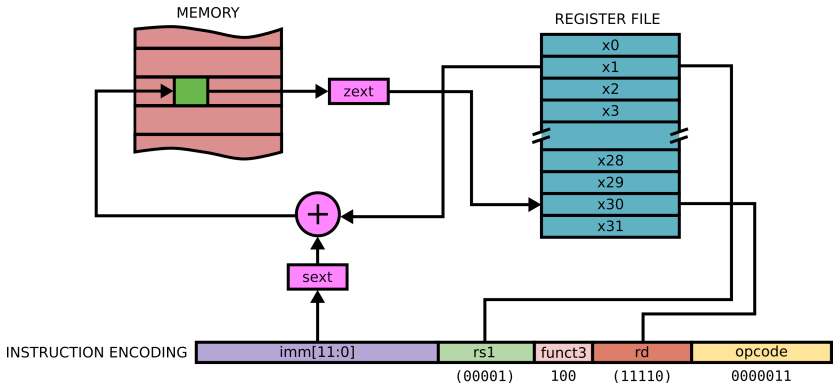
**Assembly:** `LBU rd, int12(rs1)`

**Function:** $(rd) \leftarrow zext([(rs1) + sext(imm)]_h)$

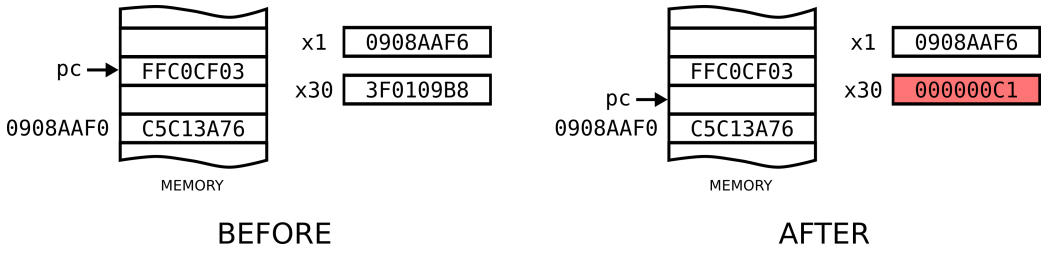
**Description:**

Adds the signed offset to the contents of register `rs1`, treating it as a memory address. Retrieves a byte from memory at that address, and zero-extends it. The result is placed in the destination register.

**Data Path:**




**Example:** `LBU x30, -4(x1)`




In this example, the little-endian layout in memory at address `0908AAF0` is `76 3A C1 C5`. The address to access is `0908AAF2`, which contains the byte `C1`. When zero-extended to 32 bits, this is `000000C1`.

# LH

Load half word

**Instruction Format:** I

**Opcode Type:** LOAD

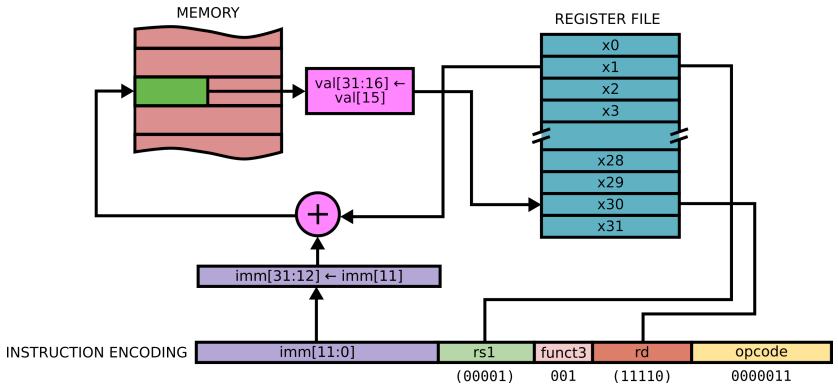
**Assembly:** `LH rd, int12(rs1)`

**Function:** `(rd) ← sext([(rs1) + sext(imm)]_h)`

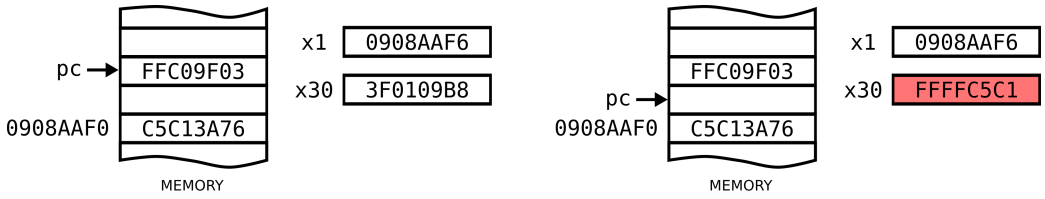
**Description:**

Adds the signed offset to the contents of register `rs1`, treating it as a memory address. Retrieves a little-endian 16-bit value from memory at that address, and sign-extends it. The result is placed in the destination register.

**Data Path:**




**Example:** `LH x30, -4(x1)`




In this example, the little-endian layout in memory at address `0908AAF0` is `76 3A C1 C5`. The address to access is `0908AAF2`, which contains the bytes `C1 C5`. The corresponding little-endian value is `C5C1`. When sign-extended to 32 bits, this is `FFFFC5C1`.

# LHU

Load half word unsigned

**Instruction Format:** I

**Opcode Type:** LOAD

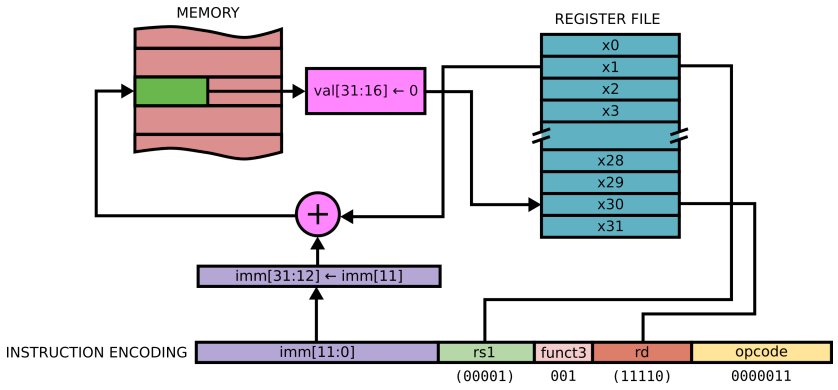
**Assembly:** `LHU rd, int12(rs1)`

**Function:** `(rd) ← zext([(rs1) + sext(imm)]h)`

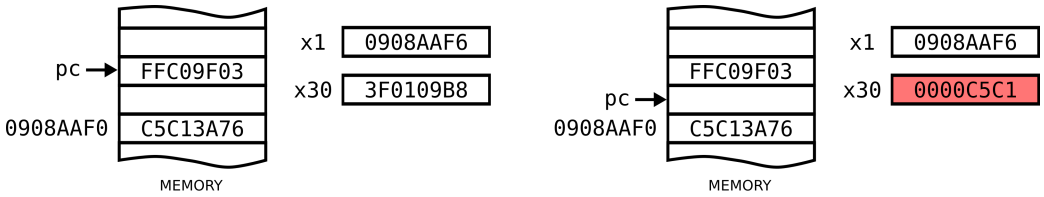
**Description:**

Adds the signed offset to the contents of register `rs1`, treating it as a memory address. Retrieves a little-endian 16-bit value from memory at that address, and zero-extends it. The result is placed in the destination register.

**Data Path:**




**Example:** `LHU x30, -4(x1)`




In this example, the little-endian layout in memory at address `0908AAF0` is `76 3A C1 C5`. The address to access is `0908AAF2`, which contains the bytes `C1 C5`. The corresponding little-endian value is `C5C1`. When zero-extended to 32 bits, this is `0000C5C1`.

# LUI

Load upper immediate

**Instruction Format:** U

**Opcode Type:** LUI

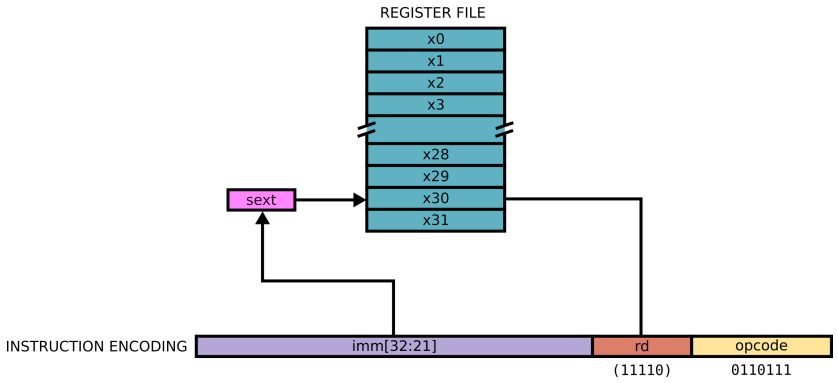
**Assembly:** `LUI rd, uint20`

**Function:** `(rd) ← sext(imm << 12)`

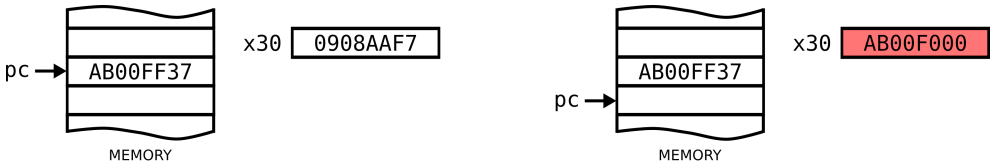
**Description:**

Constructs a 32-bit integer whose upper 20 bits are the immediate value, filling the lower 12 bits with zeros. The result is then sign-extended to XLEN bits and placed in the destination register.

**Data Path:**




**Example:** `LUI x30, 0xAB00F`




**Notes:**

The LUI instruction is used to construct constants outside the 12-bit range.

# LW

Load word

**Instruction Format:** I

**Opcode Type:** LOAD

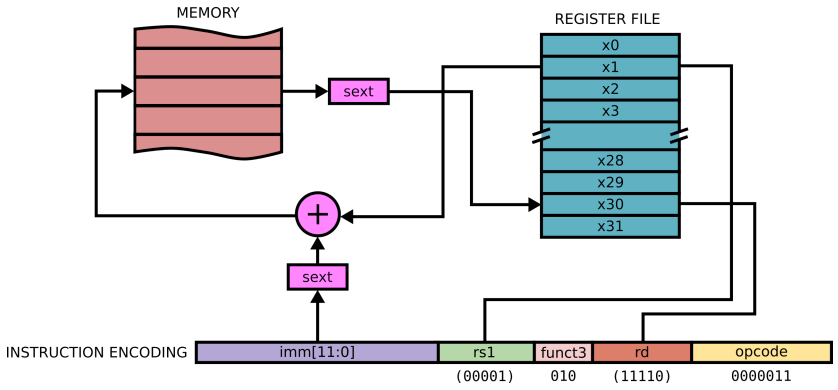
**Assembly:** `LW rd, int12(rs1)`

**Function:** `(rd) ← sext([(rs1) + sext(imm)]w)`

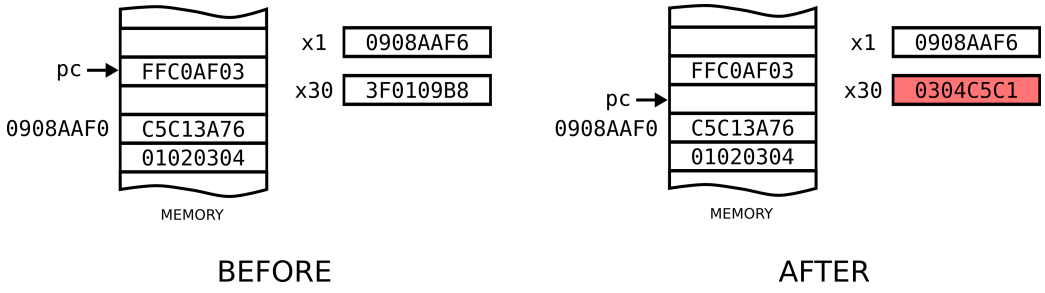
**Description:**

Adds the signed offset to the contents of register `rs1`, treating it as a memory address. Retrieves a little-endian 32-bit value from memory at that address, and sign-extends it. The result is placed in the destination register.

**Data Path:**




**Example:** `LW x30, -4(x1)`




In this example, the little-endian layout in memory at address `0908AAF0` is `76 3A C1 C5 04 03 02 01`. The address to access is `0908AAF2`, which contains the bytes `C1 C5 04 03`. The corresponding little-endian value is `0304C5C1`.

# NOP

No operation

---

**Instruction Format:** I

**Opcode Type:** LOAD

**Assembly:** *(pseudo-instruction)* `NOP`

**Function:** `(pc)` $\leftarrow$ `(pc)` $+$ `4`

**Description:**

Does not change any user-visible state, except for advancing the program counter.

**Notes:**

While there are many instructions that can do nothing, the standard `NOP` is implemented as an `ADDI` instruction:

```
ADDI x0, x0, 0
```

# OR

Bitwise or (register/register)

---

**Instruction Format:** R

**Opcode Type:** OP

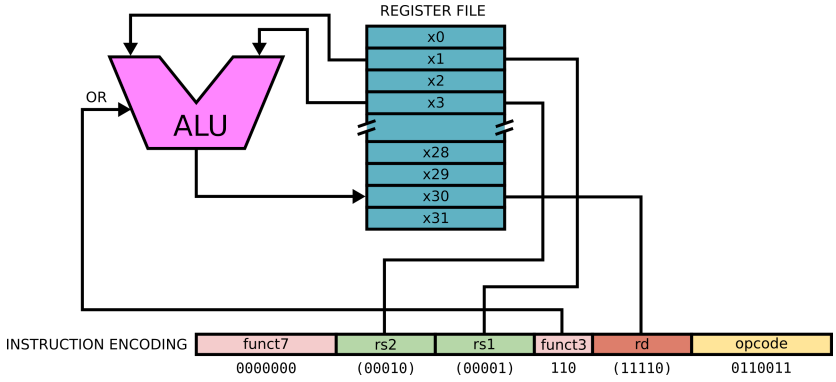
**Assembly:** `OR rd, rs1, rs2`

**Function:** `(rd) ← (rs1) ∨ (rs2)`

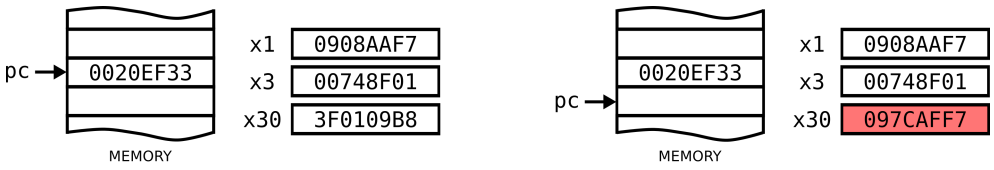
**Description:**

Performs a bitwise OR between the contents of register `rs1` and the contents of register `rs2`. The result is placed in register `rd`.

**Data Path:**




**Example:** `OR x30, x1, x3`



# ORI

Bitwise or (register/immediate)

---

**Instruction Format:** I

**Opcode Type:** OP-IMM

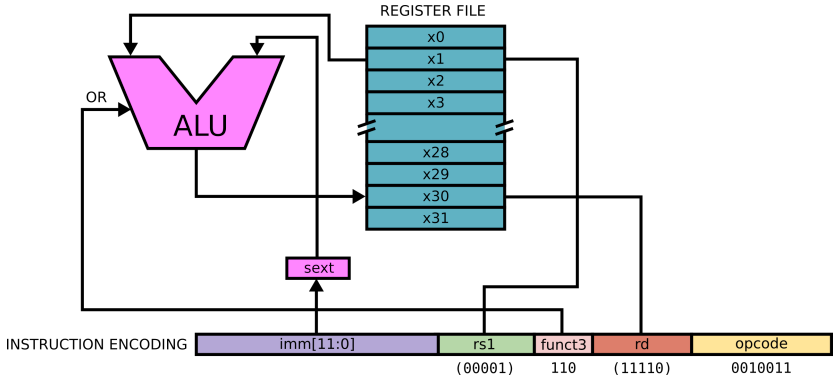
**Assembly:** `OR rd, rs1, int12`

**Function:** `(rd) ← (rs1) ∨ sext(imm)`

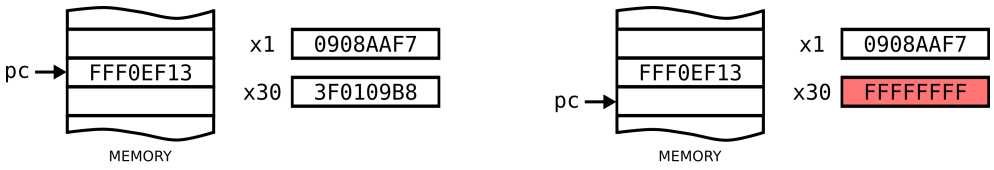
**Description:**

Performs a bitwise OR between the contents of register `rs1` and the sign-extended immediate value. The result is placed in register `rd`.

**Data Path:**




**Example:** `ORI x30, x1, -1`




BEFORE AFTER

# SB

Store byte

**Instruction Format:** S

**Opcode Type:** STORE

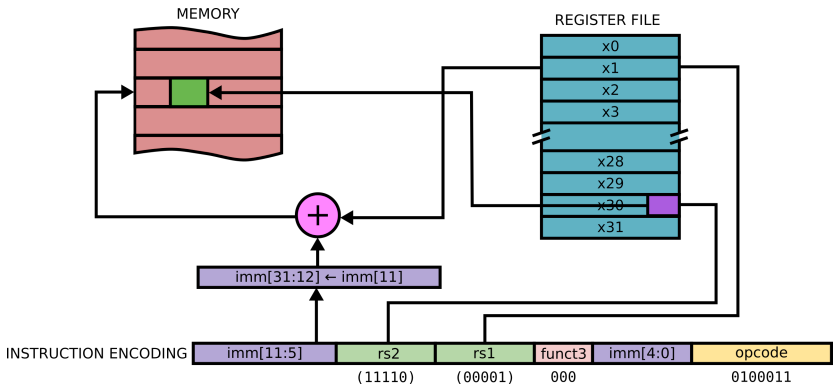
**Assembly:** `SB int12(rs1), rs2`

**Function:** $[(rs1) + sext(imm)]_b \leftarrow (rs2)_b$

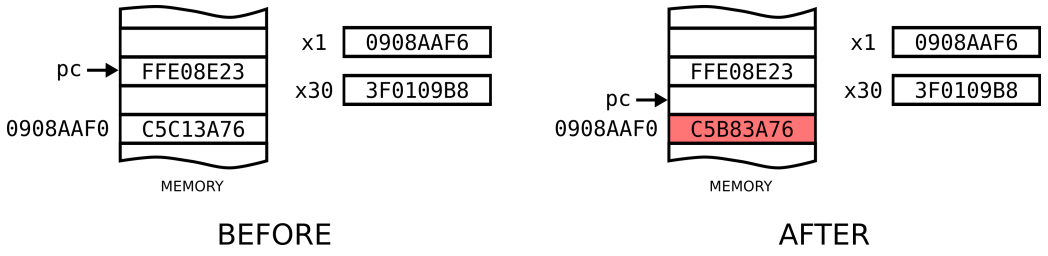
**Description:**

Adds the signed offset to the contents of register `rs1`, treating it as a memory address. Stores the bottom 8 bits of `rs2` into memory at that address.

**Data Path:**




**Example:** `SB -4(x1), x30`




In this example, the bytes in memory starting at address `0908AAF0` are `76 3A C1 C5`. The address to access is `0908AAF2`, which contains the byte `C1`. The value of the lower 8 bits in `x30` is `B8`. This is stored in memory, replacing `C1`, so that now the bytes in memory starting at address `0908AAF0` are `76 3A B8 C5`.

# SH

Store half word

**Instruction Format:** S

**Opcode Type:** STORE

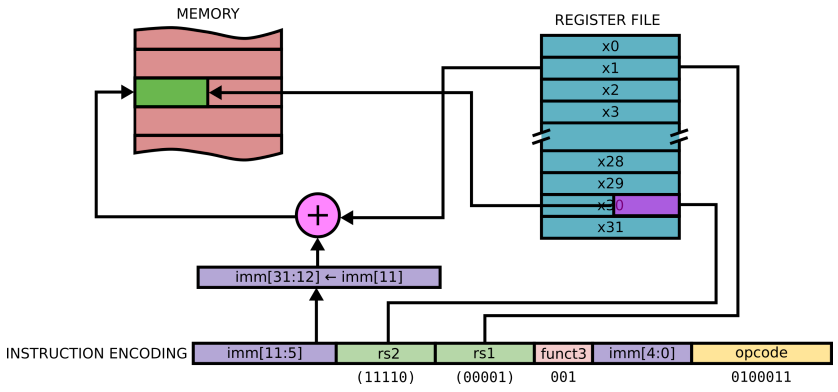
**Assembly:** `SH int12(rs1), rs2`

**Function:** $[(rs1) + sext(imm)]_h \leftarrow (rs2)_h$

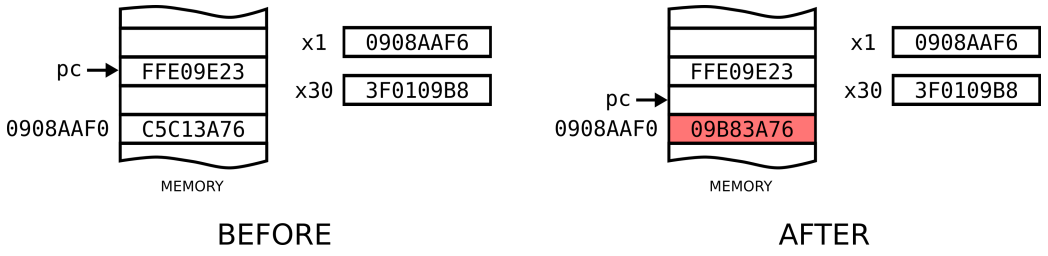
**Description:**

Adds the signed offset to the contents of register `rs1`, treating it as a memory address. Stores the bottom 16 bits of `rs2` into memory at that address.

**Data Path:**




**Example:** `SH -4(x1), x30`




In this example, the bytes in memory starting at address `0908AAF0` are `76 3A C1 C5`. The address to access is `0908AAF2`, which contains the bytes `C1 C5`. The value of the lower 8 bits in `x30` is `09B8`. This is stored in memory as the little-endina bytes `B8 09`, replacing `C1 C5`, so that now the bytes in memory starting at address `0908AAF0` are `76 3A B8 C5`.

# SLL

Shift left logical (register/register)

**Instruction Format:** R

**Opcode Type:** OP

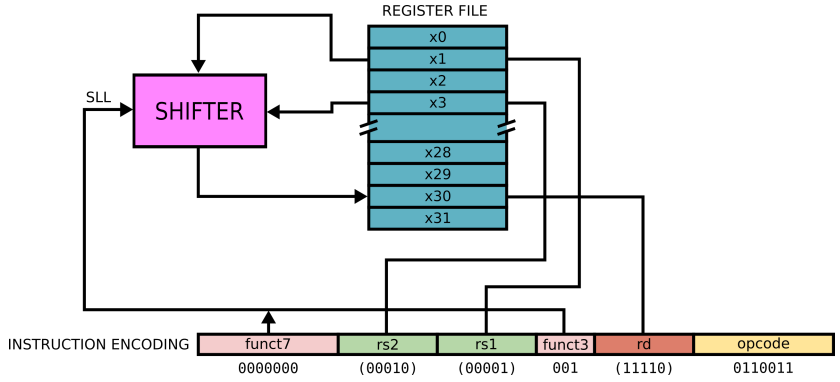
**Assembly:** `SLL rd, rs1, rs2`

**Function:** `(rd) ← (rs1) << (rs2)`$_n$

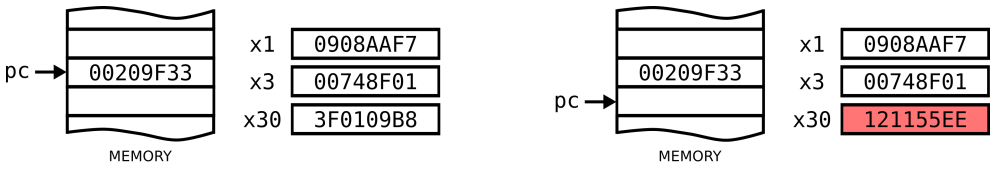
**Description:**

Shifts the contents of register `rs1` left by the lowest $\log_2$(XLEN) bits of the contents of register `rs2`, shifting in zeroes. The result is stored in the destination register.

**Data Path:**




**Example:** `SLL x30, x1, x3`



# SLLI

Shift left logical (register/immediate)

**Instruction Format:** I

**Opcode Type:** OP-IMM

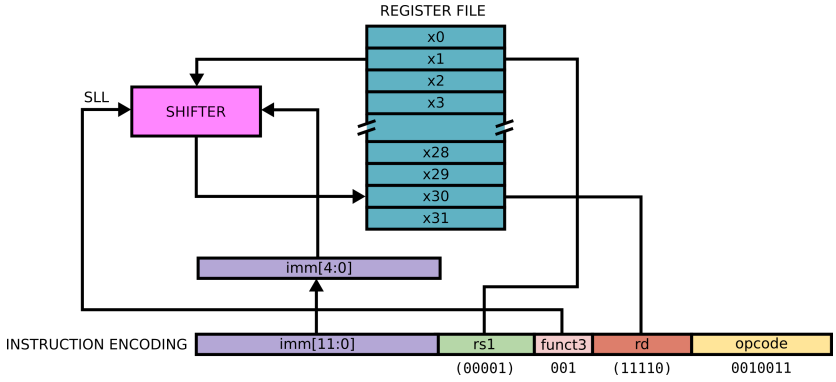
**Assembly:** `SLLI rd, rs1, uint12`

**Function:** `(rd)` $\leftarrow$ `(rs1)` $<<$ $\texttt{imm}_n$

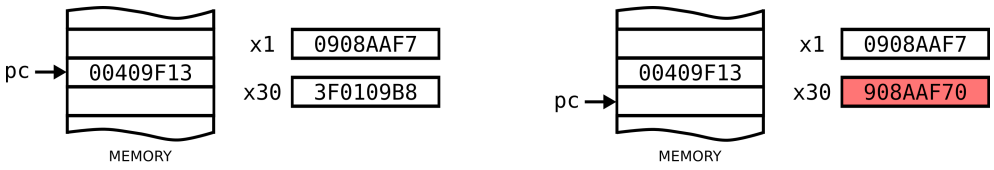
**Description:**

Shifts the contents of register `rs1` left by the lowest $\log_2$(XLEN) bits of the immediate number, shifting in zeroes. The result is stored in the destination register.

**Data Path:**




**Example:** `SLLI x30, x1, 4`




BEFORE AFTER

# SLT

Set if less than (register/register)

**Instruction Format:** R

**Opcode Type:** OP

**Assembly:** `SLT rd, rs1, rs2`

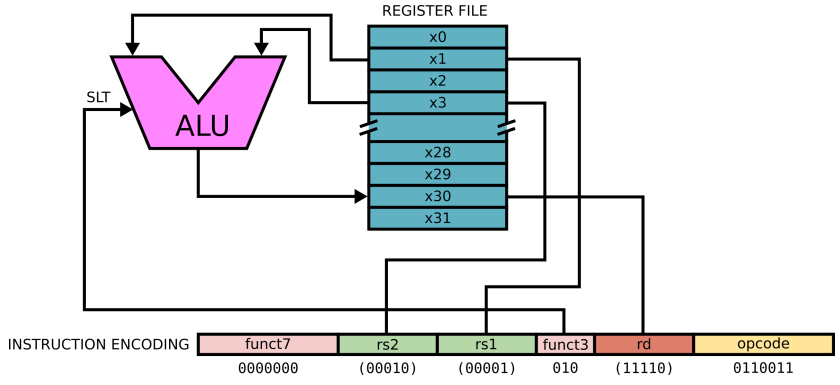
**Function:**

$$(\texttt{rd}) \leftarrow \begin{cases} \texttt{1} & \texttt{if (rs1) < (rs2)} \\ \texttt{0} & \texttt{otherwise} \end{cases}$$

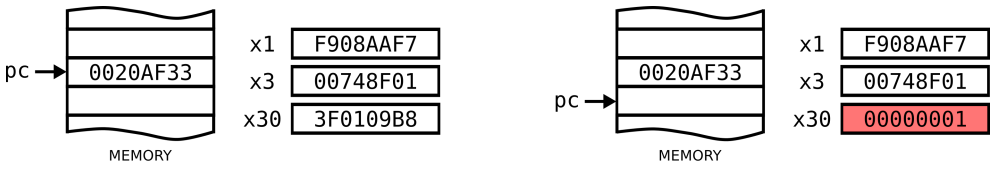
**Description:**

Sets the destination register to `1` if the contents of register `rs1` is less than the contents of register `rs2` in a signed comparison, otherwise `0`.

**Data Path:**




**Example:** `SLT x30, x1, x3`




BEFORE AFTER

# SLTI

Set if less than (register/immediate)

**Instruction Format:** I

**Opcode Type:** OP-IMM

**Assembly:** `SLTI rd, rs1, int12`

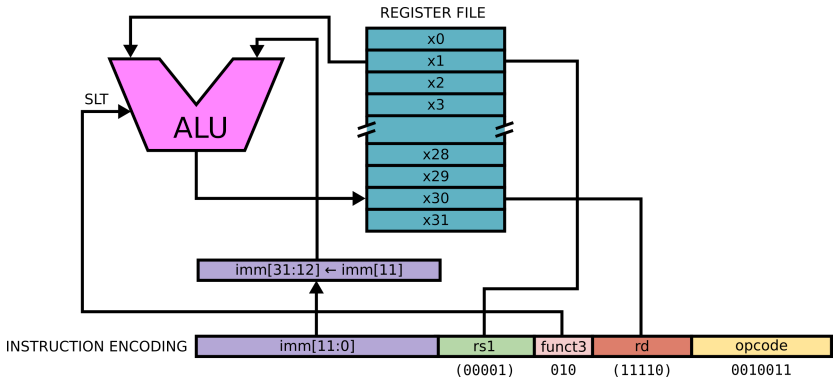
**Function:**

$$(\texttt{rd}) \leftarrow \begin{cases} \texttt{1} & \texttt{if (rs1) < sext(imm)} \\ \texttt{0} & \texttt{otherwise} \end{cases}$$

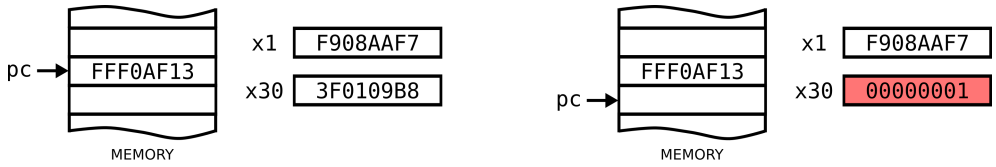
**Description:**

Sets the destination register to 1 if the contents of register `rs1` is less than the sign-extended immediate value in a signed comparison, otherwise `0`.

**Data Path:**




**Example:** `SLTI x30, x1, -1`




BEFORE AFTER

**Notes:**

Overflows of the `ADD` instruction for signed arithmetic can be checked using `SLTI` and `SLT`:

```
ADD   x1, x2, x3
SLTI  x4, x3, 0
SLT   x5, x1, x2
BNE   x4, x5, overflow
```

# SLTIU — Set if less than, unsigned (register/immediate)

**Instruction Format:** I

**Opcode Type:** OP-IMM

**Assembly:** `SLTIU rd, rs1, uint12`

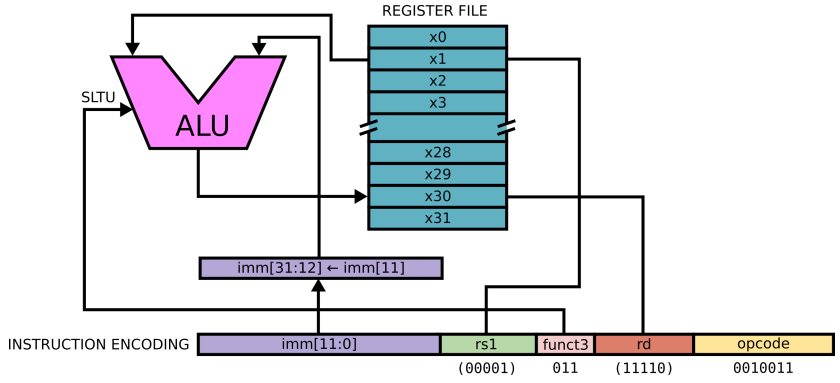
**Function:**

$$(\texttt{rd}) \leftarrow \begin{cases} 1 & \text{if } (\texttt{rs1})_u < \texttt{zext(imm)} \\ 0 & \text{otherwise} \end{cases}$$

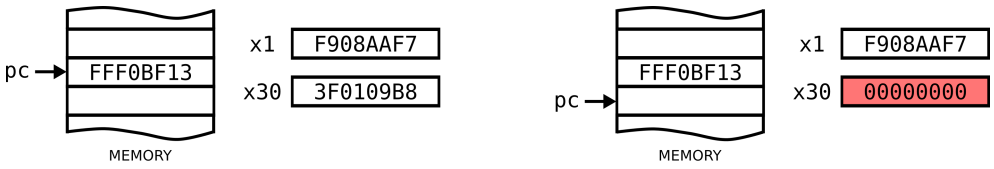
**Description:**

Sets the destination register to 1 if the contents of register `rs1` is less than the zero-extended immediate value in an unsigned comparison, otherwise `0`.

**Data Path:**




**Example:** `SLTIU x30, x1, 0xFFF`




BEFORE AFTER

# SLTU — Set if less than, unsigned (register/register)

**Instruction Format:** R

**Opcode Type:** OP

**Assembly:** `SLTU rd, rs1, rs2`

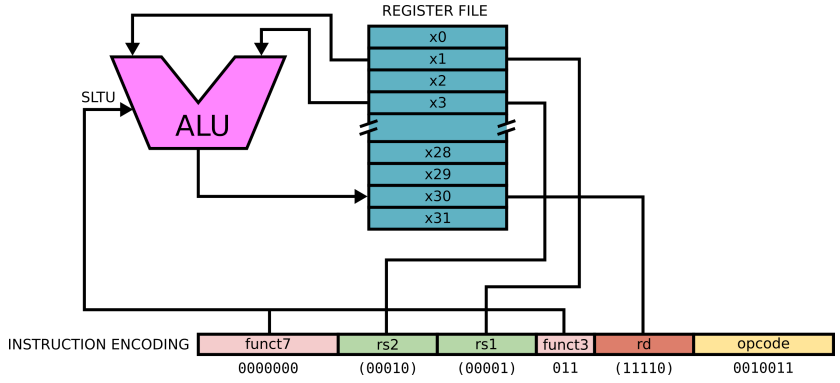
**Function:**

$$(\texttt{rd}) \leftarrow \begin{cases} 1 & \text{if } (\texttt{rs1})_u < (\texttt{rs2})_u \\ 0 & \text{otherwise} \end{cases}$$

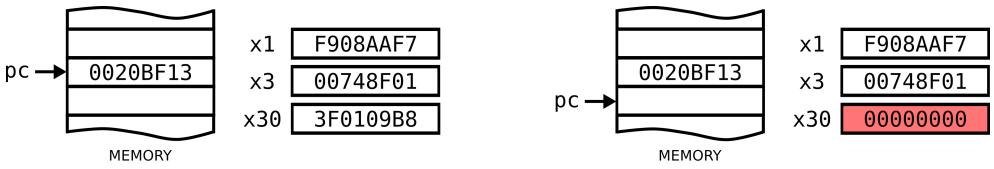
**Description:**

Sets the destination register to `1` if the contents of register `rs1` is less than the contents of register `rs2` in an unsigned comparison, otherwise `0`.

**Data Path:**




**Example:** `SLTU x30, x1, x3`




BEFORE AFTER

# SRA

Shift right arithmetic (register/register)

**Instruction Format:** R

**Opcode Type:** OP

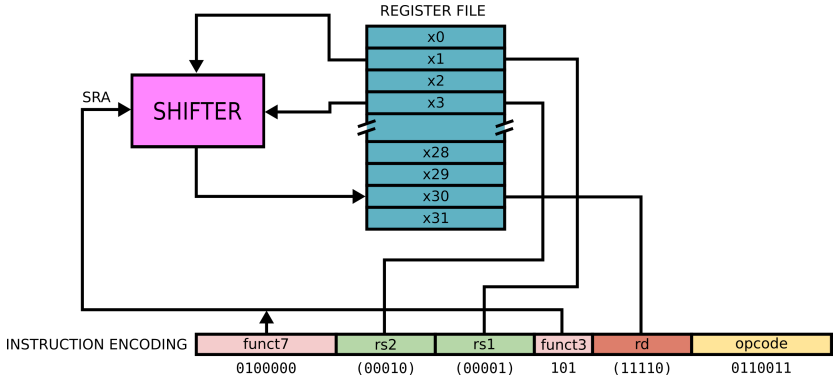
**Assembly:** `SRA rd, rs1, rs2`

**Function:** `(rd)` $\leftarrow$ `(rs1)` $>>$ `(rs2)`$_n$

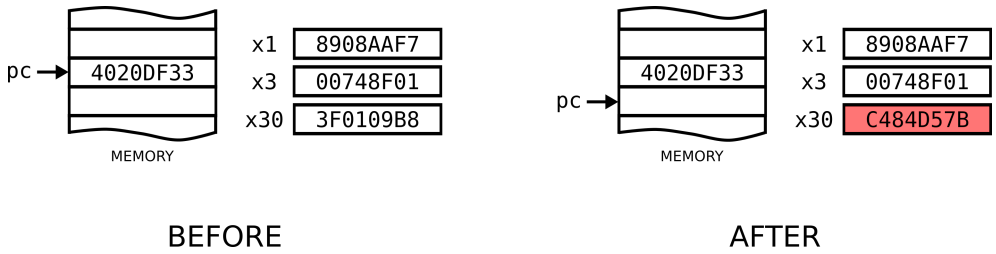
**Description:**

Shifts the contents of register `rs1` right by the lowest $\log_2$(XLEN) bits of the contents of register `rs2`, shifting in the most significant bit of register `rs1`. The result is stored in the destination register.

**Data Path:**




**Example:** `SRA x30, x1, x3`



# SRAI

Shift right arithmetic (register/immediate)

**Instruction Format:** I

**Opcode Type:** OP-IMM

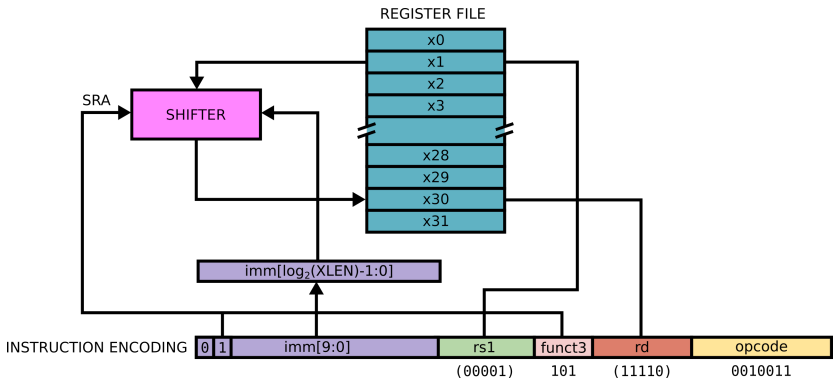
**Assembly:** `SRAI rd, rs1, uintN`

**Function:** `(rd)` $\leftarrow$ `(rs1)` $>>$ `(imm)`$_n$

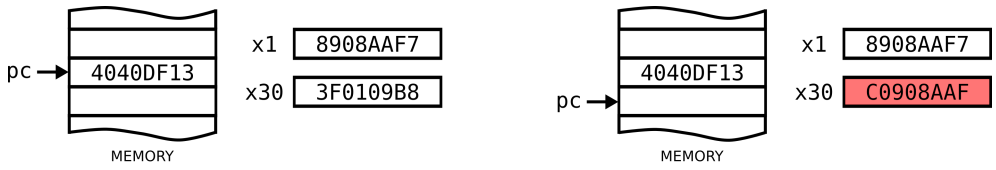
**Description:**

Shifts the contents of register `rs1` right by the lowest $\log_2$(XLEN) bits of the immediate value, shifting in the most significant bit of register `rs1`. The result is stored in the destination register.

**Data Path:**




**Example:** `SRAI x30, x1, 4`



# SRL

Shift right logical (register/register)

**Instruction Format:** R

**Opcode Type:** OP

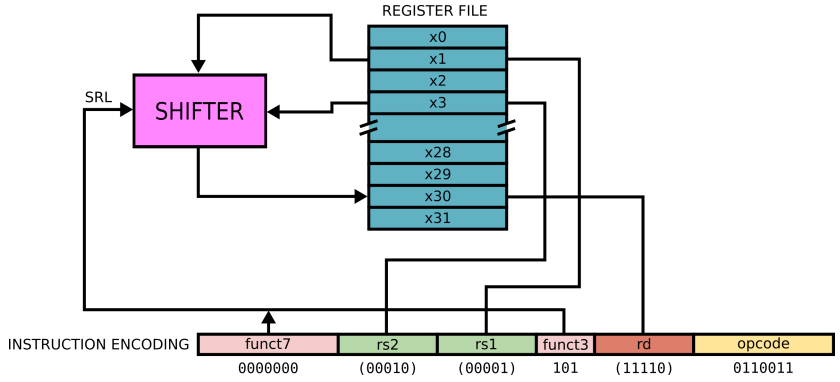
**Assembly:** `SRL rd, rs1, rs2`

**Function:** `(rd)` $\leftarrow$ `(rs1)` $>>>$ `(rs2)`$_n$

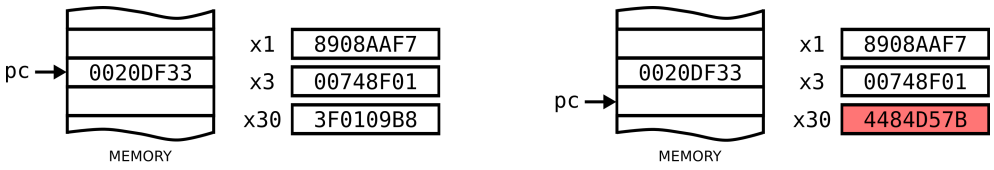
**Description:**

Shifts the contents of register `rs1` right by the lowest $\log_2$(XLEN) bits of the contents of register `rs2`, shifting in zeroes. The result is stored in the destination register.

**Data Path:**




**Example:** `SRL x30, x1, x3`



# SRLI

Shift right logical (register/immediate)

**Instruction Format:** I

**Opcode Type:** OP-IMM

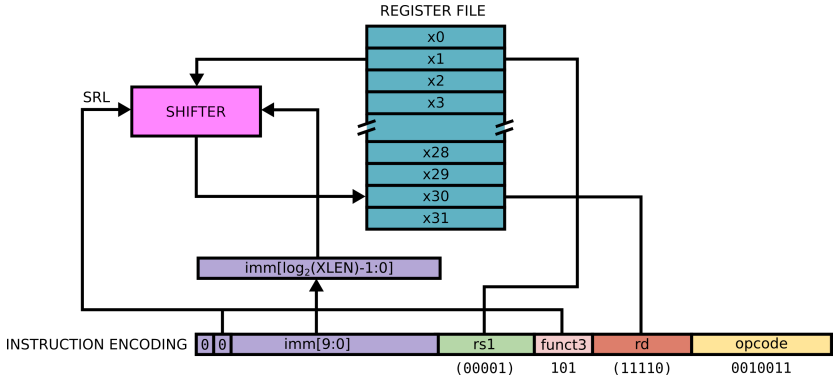
**Assembly:** `SRLI rd, rs1, uintN`

**Function:** `(rd) ← (rs1) >>> (imm)`$_n$

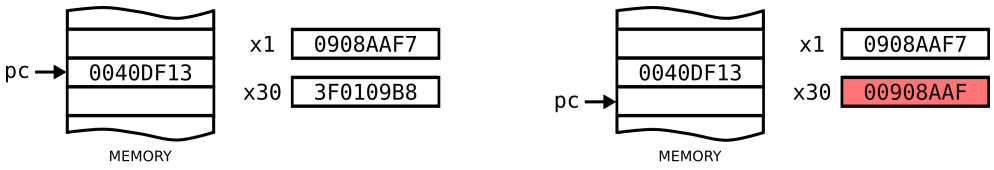
**Description:**

Shifts the contents of register `rs1` right by the lowest $\log_2$(XLEN) bits of the immediate value, shifting in zeroes. The result is stored in the destination register.

**Data Path:**




**Example:** `SRLI x30, x1, 4`



# SUB

Subtract (register/register)

**Instruction Format:** R

**Opcode Type:** OP

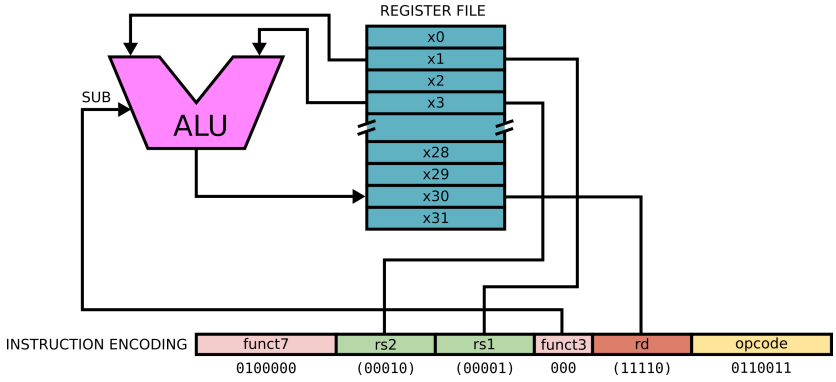
**Assembly:** `SUB rd, rs1, rs2`

**Function:** `(rd)` $\leftarrow$ `(rs1)` $-$ `(rs2)`

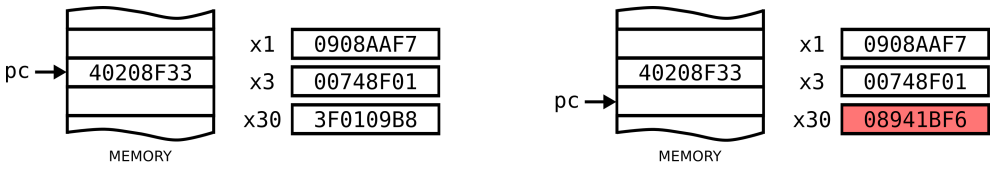
**Description:**

Subtracts contents of register `rs2` from the contents of register `rs1`. The result is stored in the destination register.

**Data Path:**




**Example:** `SUB x30, x1, x3`




BEFORE AFTER

**Notes:**

There is no separate `SUBI` instruction. However, `SUBI` is a psuedo-instruction encoded as `ADDI` where the immediate value is negated. Thus, the following two instructions are encoded the same way:

```
SUBI  x30, x1, 1
ADDI  x30, x1, -1
```

# SW

Store word

**Instruction Format:** S

**Opcode Type:** STORE

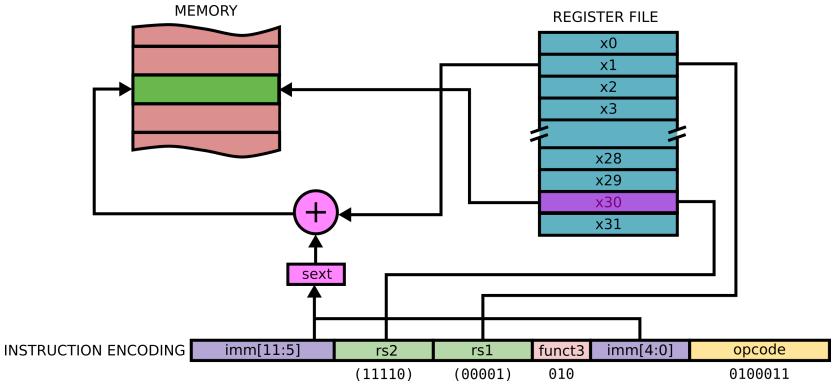
**Assembly:** `SW int12(rs1), rs2`

**Function:** $[(rs1) + sext(imm)]_w \leftarrow (rs2)_w$

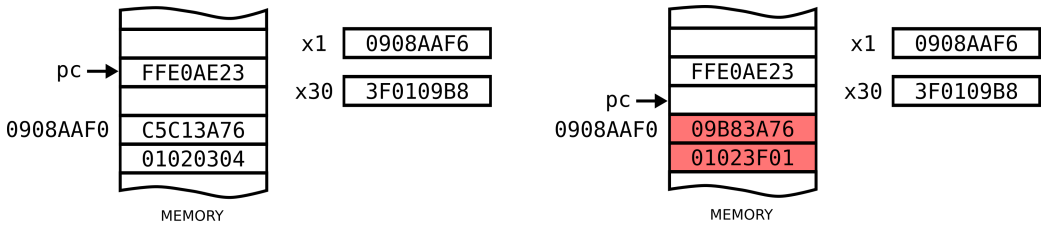
**Description:**

Adds the signed offset to the contents of register `rs1`, treating it as a memory address. Stores the bottom 32 bits of `rs2` into memory at that address.

**Data Path:**




**Example:** `SW -4(x1), x30`




In this example, the bytes in memory starting at address `0908AAF0` are `76 3A C1 C5 04 03 02 01`. The address to access is `0908AAF2`, which contains the bytes `C1 C5 04 03`. The contents of x30 is `3F0109B8`. This is stored in memory as the little-endian bytes `B8 09 01 3F`, replacing `C1 C5 04 03`, so that now the bytes in memory starting at address `0908AAF0` are `76 3A B8 09 01 3F 02 01`.

# XOR

Bitwise exclusive or (register/register)

**Instruction Format:** R

**Opcode Type:** OP

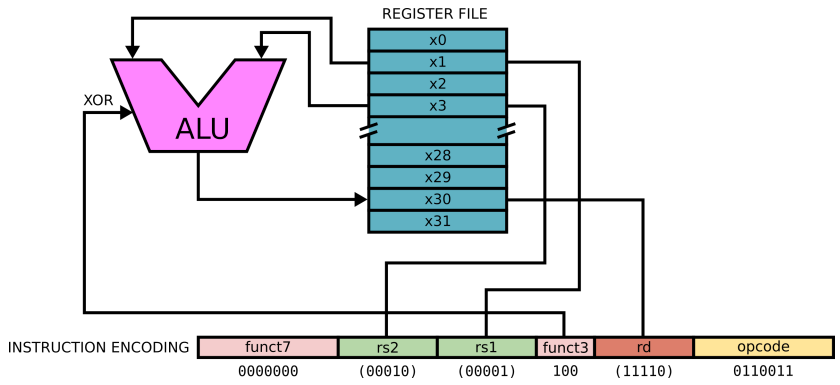
**Assembly:** `XOR rd, rs1, rs2`

**Function:** `(rd)` $\leftarrow$ `(rs1)` $\oplus$ `(rs2)`

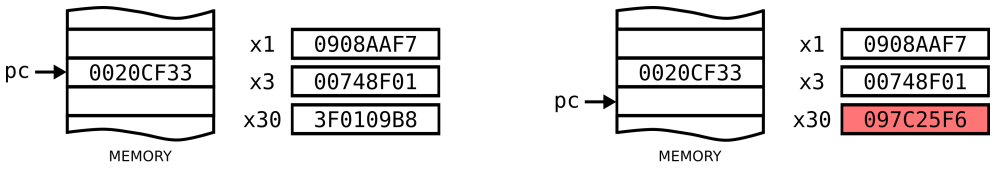
**Description:**

Performs a bitwise XOR between the contents of register `rs1` and the contents of register `rs2`. The result is placed in register `rd`.

**Data Path:**




**Example:** `XOR x30, x1, x3`




BEFORE AFTER

# XORI

Bitwise exclusive or (register/immediate)

**Instruction Format:** I

**Opcode Type:** OP-IMM

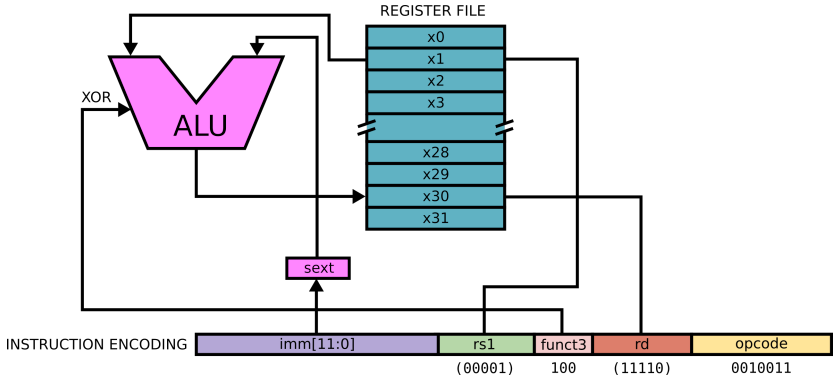
**Assembly:** `XOR rd, rs1, int12`

**Function:** `(rd)` $\leftarrow$ `(rs1)` $\oplus$ `sext(imm)`

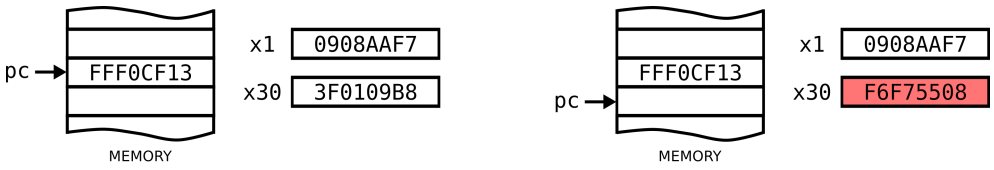
**Description:**

Performs a bitwise XOR between the contents of register `rs1` and the sign-extended immediate value. The result is placed in register `rd`.

**Data Path:**




**Example:** `XORI x30, x1, -1`




BEFORE AFTER