HEIRV32

32-bit, non-minimal ISA, soft-processor

Hes·so VALAIS WALLIS

School of Engineering

HEIRV32 is an (under-minimal ***Instruction Set Architecture***) implementation of the RISC-V microprocessor architecture, intended for teaching.

It is developed and implemented during the **C**omputer **Ar**chitecture course. Both a ***single-cycle*** and a ***multi-cycle*** versions are available.

It supports basic load (***lw***), store (***sw***), R-type (***add, sub, or, and, slt***), I-type (***addi, ori, andi, slti***), branch (***beq***) and jump (***jal***) instructions.

A dedicated assembler allows for readable codes to be translated to a BRAM file which can be used by HEIRV32 to work.

Registers

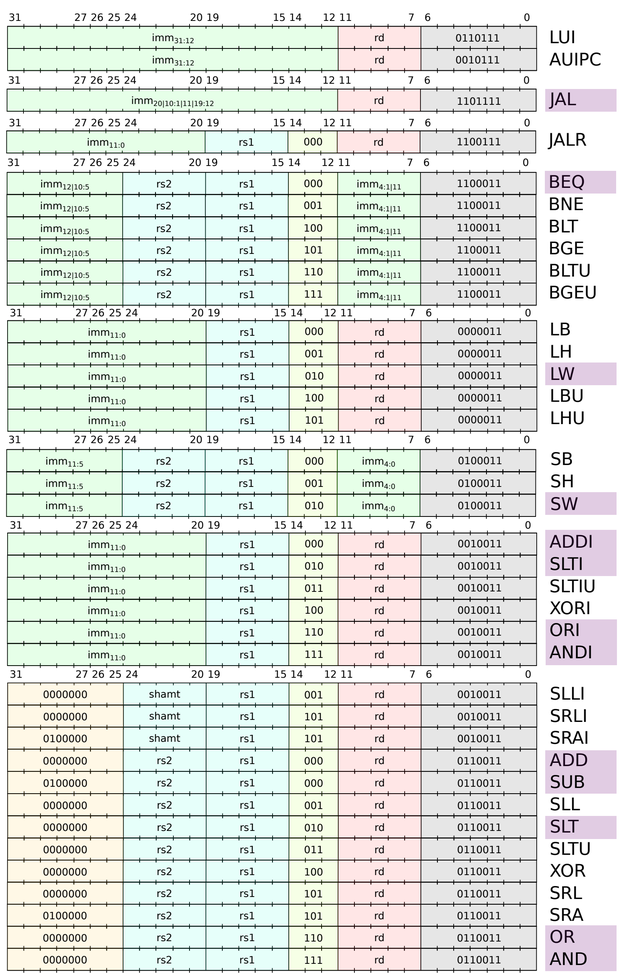
Register	ABI Name	Description	Saver
x0	zero	Zero constant	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5-x7	t0-t2	Temporaries	Caller
x8	s0 / fp	Saved / frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Fn args/return values	Caller
x12-x17	a2-a7	Fn args	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller
f0-7	ft0-7	FP temporaries	Caller
f8-9	fs0-1	FP saved registers	Callee
f10-11	fa0-1	FP args/return values	Caller
f12-17	fa2-7	FP args	Caller
f18-27	fs2-11	FP saved registers	Callee
f28-31	ft8-11	FP temporaries	Caller
x30	leds	Leds wr source	Caller
x31	btns	Buttons rd source	External

Official ISA Sets

I	Integer base instructions
M	Math, Integer multiplication and division instructions
A	Atomic instructions
F	Single-precision floating-point instructions
D	Double-precision floating-point instructions
G	General (combining I + M + A + F + D)
Q	Quad-precision floating-point instructions
L	Decimal floating point instructions
C	Compressed instructions
B	Bit manipulation instructions
J	Dynamically translated languages
T	Transactional memory instructions
P	Packed-SIMD instructions
V	Vector operations instructions
N	User-level interrupt instructions

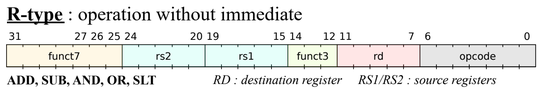
HEIRV32 does not meet the minimal I set.

RV32I Base Instructions



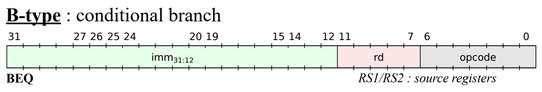
HEIRV32 Instruction Set

R-type : operation without immediate



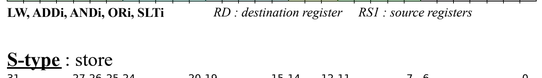
ADD, SUB, AND, OR, SLT *RD : destination register RS1/RS2 : source registers*

B-type : conditional branch



BEQ *RS1/RS2 : source registers*

I-type : operation with immediate



LW, ADDi, ANDi, ORi, SLTi *RD : destination register RS1 : source registers*

U-type : 20-bits immediate

None *RD : destination register*

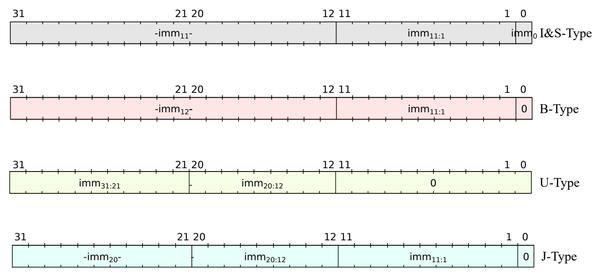
S-type : store

SW *RD : destination register RS1/RS2 : source registers*

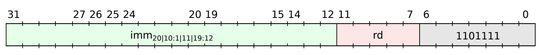
J-type : unconditional branch

JAL *RD : destination register*

Immediates Distribution

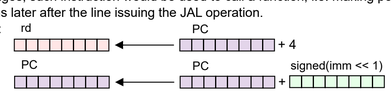


JAL

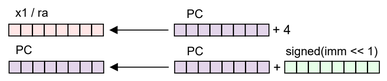


The Jump And Link instruction is a relative jump (i.e. adds a signed offset to the current PC), while registering the current PC+4 into a given register (default to x1/ra - return address).
In higher level languages, such instruction would be used to call a function, i.e. making possible to resume the operations later after the line issuing the JAL operation.
In ASM one can use :

jal rd imm

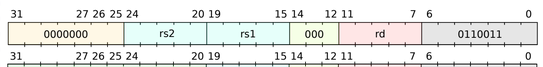


jal imm

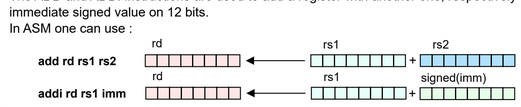


The immediate value is normally given on 20 bits, but without support for U-type instructions, its value is limited to 12 bits, which will be multiplied by two internally (no access to odd addresses).

ADD, ADDI

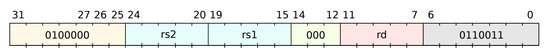


The ADD and ADDI instructions are used to add a register with another one, respectively an immediate signed value on 12 bits.
In ASM one can use :

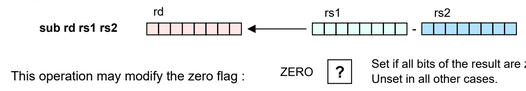


This operation may modify the zero flag : ZERO ? Set if all bits of the result are zero. Unset in all other cases.

SUB

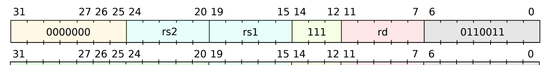


The SUB instruction is used to substract two registers from each other.
There is no immediate equivalent of the instruction, result which can obtained by using ADDI with a negative value.
In ASM one can use :

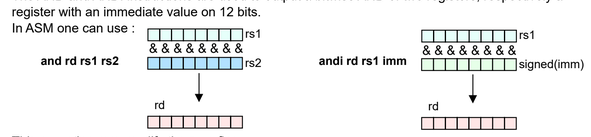


This operation may modify the zero flag : ZERO ? Set if all bits of the result are zero. Unset in all other cases.

AND, ANDI

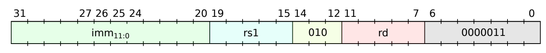


The AND and ANDI instructions are used to output a bitwise AND of two registers, respectively a register with an immediate value on 12 bits.
In ASM one can use :

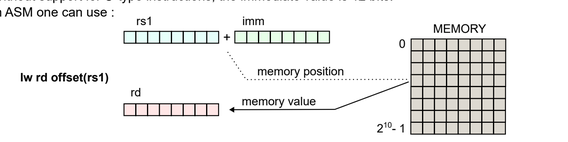


This operation may modify the zero flag.

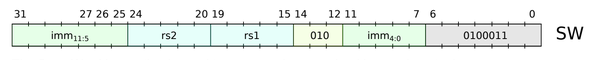
LW



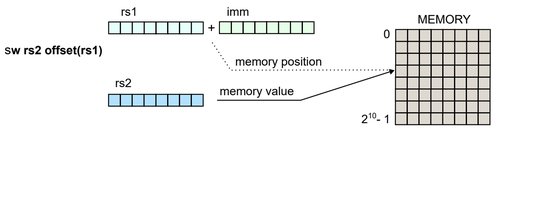
The Load Word instruction is used to fetch a 32 bits value directly from the memory and save it inside a register.
Without support for U-type instructions, the immediate value is 12 bits.
In ASM one can use :



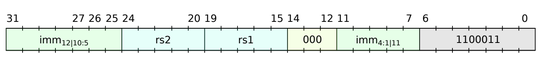
SW



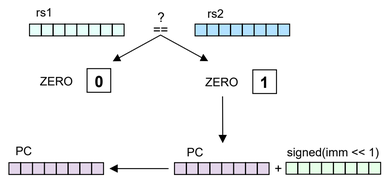
The Save Word instruction is used to save a value contained in a register to the memory.
Without support for U-type instructions, the immediate value is 12 bits.
In ASM one can use :



BEQ

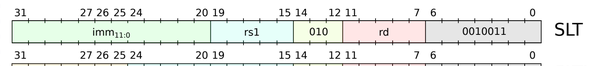


The BEQ instruction is a relative jump (i.e. adds a signed offset to the current PC) which triggers only if two registers values are strictly equivalent.
It *DOES NOT* register the current PC value.
In higher level languages, such instruction would be used as conditional branching (if, loops etc.).
In ASM one can use :

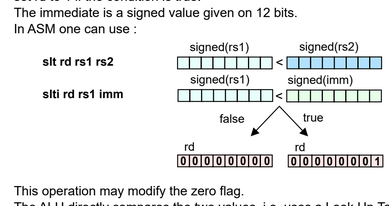


The ALU does not contain a direct comparison system. To check for equality, the ALU will calculate rs1 - rs2, and check if the ZERO flag is set. If so, the PC is updated.

SLT, SLTI



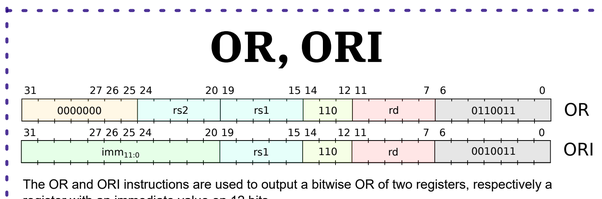
The SLT and SLTI instructions check if signed(rs1) < signed(rs2) (respectively signed(imm)) and set rd to 1 if the condition is true.
The immediate is a signed value given on 12 bits.
In ASM one can use :



This operation may modify the zero flag.
The ALU directly compares the two values, i.e. uses a Look Up Table for this purpose.

OR, ORI

The OR and ORI instructions are used to output a bitwise OR of two registers, respectively a register with an immediate value on 12 bits.
In ASM one can use :



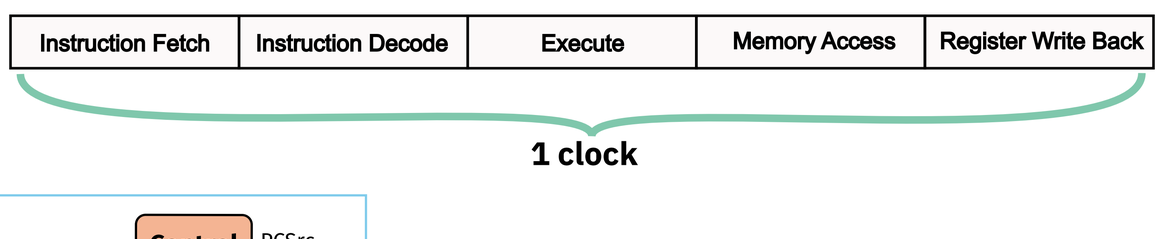
This operation may modify the zero flag.

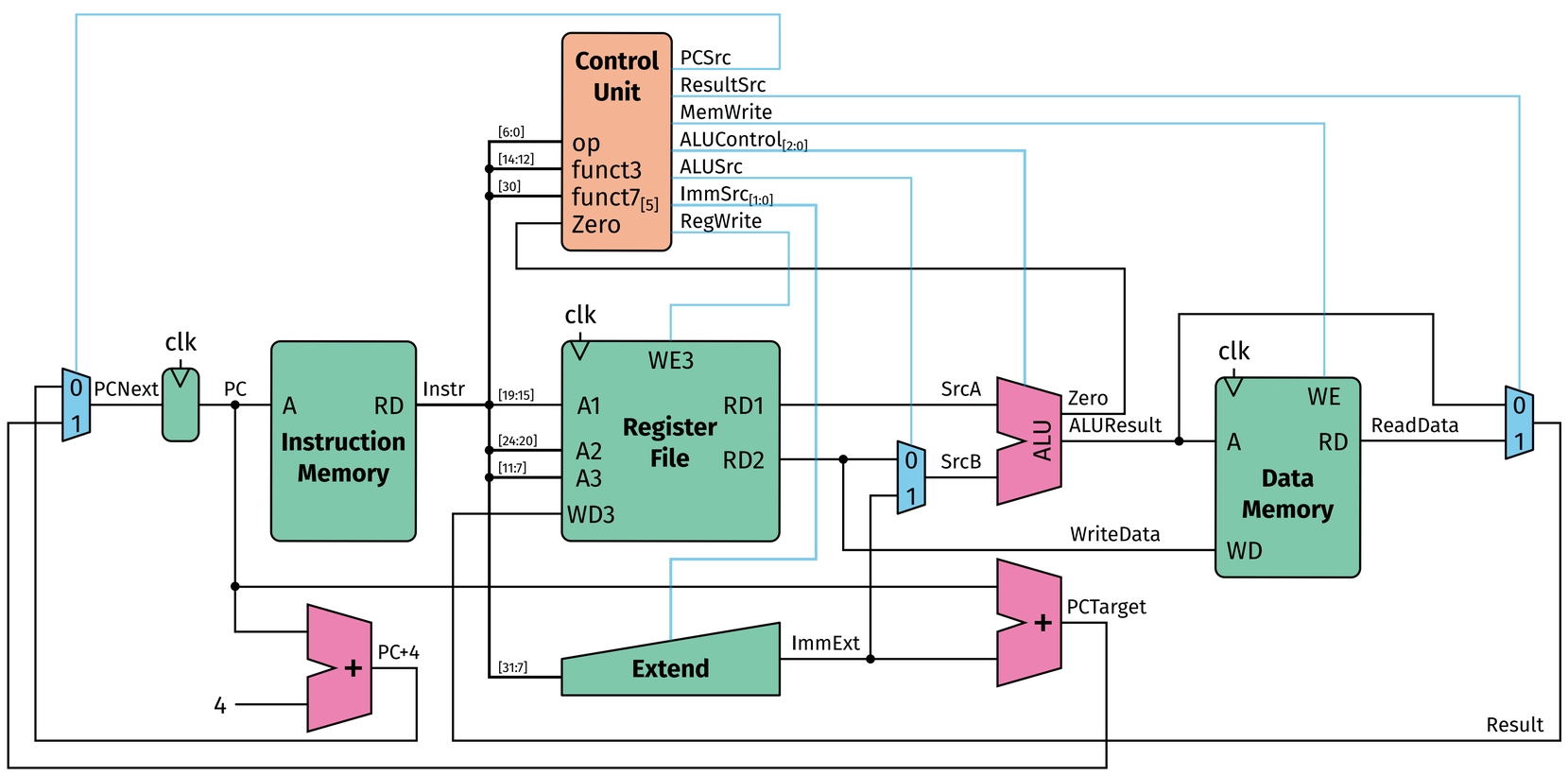
Architecture

HEIRV32 is available in two architectures :

The **multi-cycle** architecture cuts down an instruction in three to five clock cycles. With such implementation the memory is synchronous, thus flashs and ram blocks can be used. Faster clock speed can be considered, but pipelining is required to achieve better performances.

The **single-cycle** architecture is easier to begin with, such that all instructions take only one clock cycle to execute.
The cons, however, are slower clock speeds (longer path for the data), and asynchronous memory access (takes a lot of place in an FPGA !).
This architecture is the following :



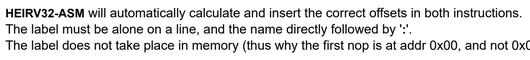


Pseudo-instructions

Pseudo-instructions are instructions created by the programmers to simplify the assembler code they write.
They are aliases for real instructions with specific predefined settings.
Those DO NOT exist inside the CPU. They are valid only when used with an assembler which supports them.
For Heirv32, **HEIRV32-ASM** allow one to write assembler code and then translate it to a BRAM-readable format (the memory/instruction part of the CPU). It also produces a binary file, which can be analyzed with specific decompilers (e.g. Ghidra).

NOP

The NOP pseudo-instruction is meant to do nothing, i.e. lose a clock period (e.g. to implement waitings).
While it is not possible to "do nothing" (except by stopping the clock and thus the whole system), the idea is to perform an operation which does not change the state of the system.
There are multiple possibilities for this.
In ASM one can use :



As seen above, the **nop** instruction is replaced by an **addi x0 x0 0** instruction when using the **HEIRV32-ASM**, which tries to add 0 with x0 and register the result in x0, which is already a special register tied to 0's and that cannot be changed.

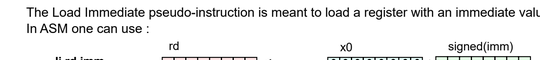
MV

The move pseudo-instruction is meant to move a register to another register.
In ASM one can use :

As seen above, the **mv** instruction is replaced by an **addi rd rs1 0** instruction when using the **HEIRV32-ASM**, which simply adds 0 with rs1 (and so does not modify its value) and register the result in rd.

LI

The Load Immediate pseudo-instruction is meant to load a register with an immediate value.
In ASM one can use :



As seen above, the **li** instruction is replaced by an **addi rd x0 imm** instruction when using the **HEIRV32-ASM**, which simply adds the immediate with x0 (i.e. 0) and register the result in rd.

Labels

The labels, while not technically instructions, are a way to simplify the offset calculation for the **beq** and **jal** instructions.
In ASM one can write its code like :

```
Addr.   myLabel:
0x00      nop
0x04      nop
0x08      beq x0 x1 myLabel     -- the calculated offset will be 0xFFC, i.e. (-8 >> 1) = -4 in 12 bits signed
0x0C      jal myLabel           -- the calculated offset will be 0xFFA, i.e. (-12 >> 1) = -6 in 12 bits signed
```

HEIRV32-ASM will automatically calculate and insert the correct offsets in both instructions.
The label must be alone on a line, and the name directly followed by ':'.
The label does not take place in memory (thus why the first nop is at addr 0x00, and not 0x04).