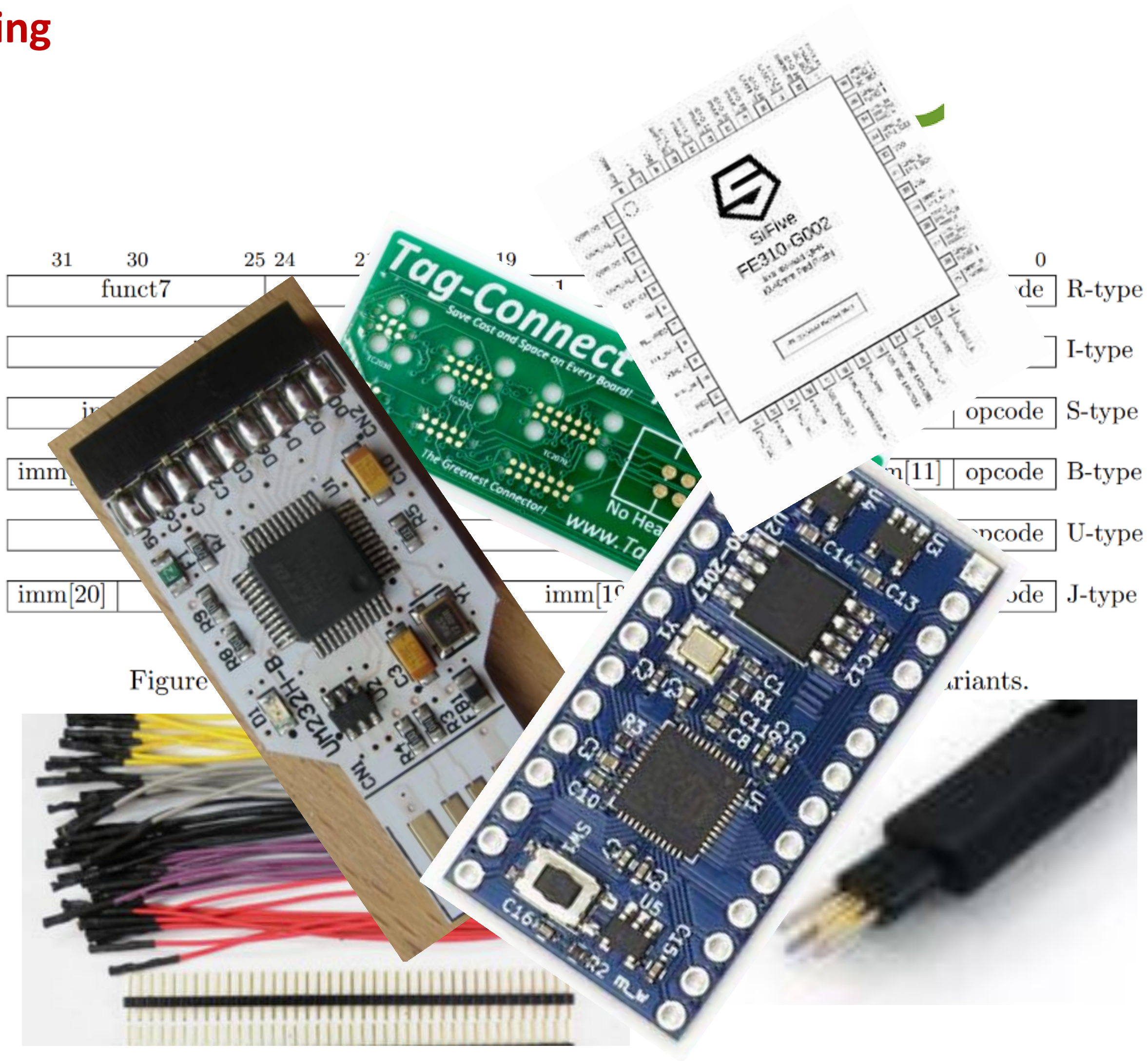
20th International Joint Conference on Computer Science and Software Engineering
28th June – 1st July 2023
Naresuan University
Pitsanulok, THAILAND

"NU4YOU" RISC-V HANDS-ON WORKSHOP



Naruemon Rattanakunakorn
Paul Sherman

Overview

- Thinking General
- What is RISC-V?
- Grammar, Constants, Variables, and Operations
- Code and Lab Etiquette
- Wiring, Assembling, Compiling, Linking, Loading, Running, and Looking
- Six Tables, Six RISC-V ISA Extensions, and have some fun
- Review and Wrap-up

Famous Abstractions In History

นามธรรม รูปธรรม

- **G. W. F. Hegel gave us the *Dialectic Method*** ***1850s***
  unity of opposites, Negation of negation, quality versus quantity
- **Max Wertheimer gave us *Gestalt Psychology*** ***1910***
- **Feynman and Schwinger gave us the *Principle of Least Action*** ***1948***
- **Christopher Booker gave us *Story Telling*** ***1970***
  Obstacles, Rags to Riches, Quest, Voyage & Return, Comedy, Tragedy, Rebirth
- **Ian Holland gave us the *Principle of Least Knowledge*** **1987**
  Law of Demeter, an object-oriented rule of style – Brad Appleton's "Big Mac"
- **Howard Sherman gives us the *Progressive Method*** ***2006***
  as applied to Politics, Economy, Technology, Ideology
- **Now, Andrew Waterman and David Patterson give us *RISC-V*** **2010**
  R-type, I-type, S-type, B-type, U-type, J-type

What is RISC-V?

- Reduced Instruction Set Computing
- Created at U.C. Berkeley to help people learn CPU design *and put into public domain for all to freely use and revise*
- An “ISA” just like Intel x86, ARM, MIPS, VAX, Power PC, ...
- No more *instructions* than you’ll ever need – only 82 of them
- More *registers* than you’ll ever need – 32 (less one)

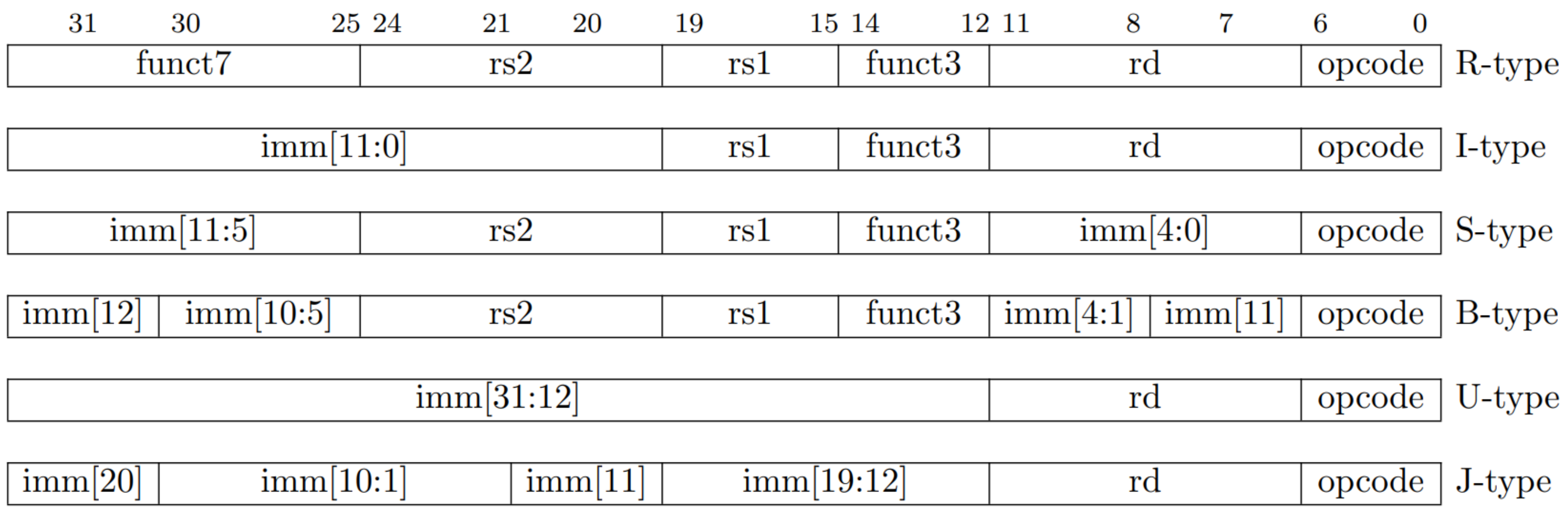
Instructions are like functions, methods, or verbs

Instruction
Set
Architecture
สถาปัตย์“กรรม”

Registers are like variables, objects, or nouns

The RISC-V “ISA”

Instruction *formats* – only 6 of them



31	30 25	24 21	20	19 15	14 12	11 8	7	6 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]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	B-type
imm[31:12]						rd		opcode	U-type
imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode	J-type

Figure 2.3: RISC-V base instruction formats showing immediate variants.

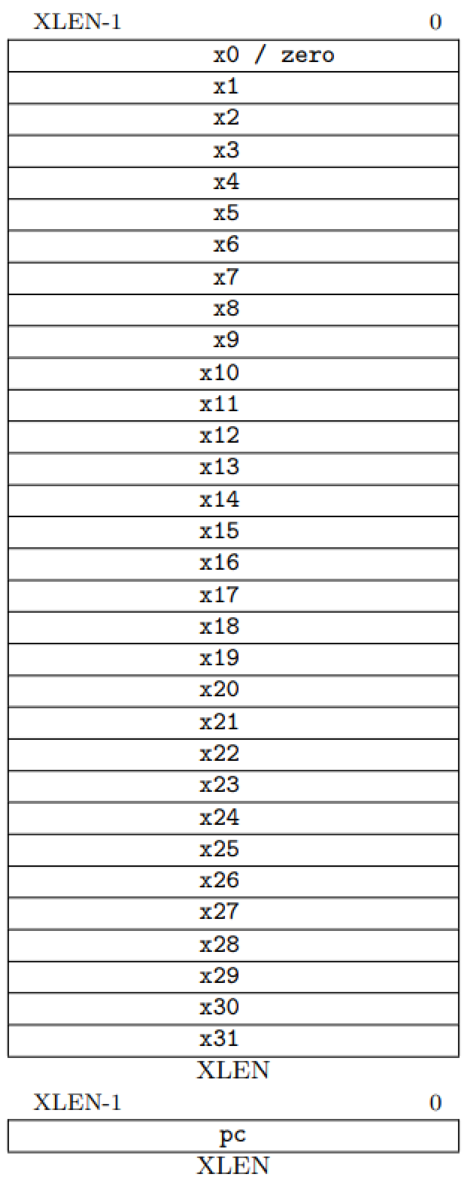


Figure 2.1: RISC-V base unprivileged integer register state.

Register set

- ✓ **All ~82 instructions can be made with these six forms**
- ✓ **Free to use any register for any purpose**
- **? Are S and B the same? Are U and J the same?**

Constant Values

- *Constants* are built into instruction words in two ways
  Short Immediate – 12-bits, the I-type forms
  Long Immediate – 20-bits, the U- and J-type forms

ขยายศูนย์

- Some constants are *Zero-extended*
- Other constants are *Sign-extended*

```
 123 = 0000 0111 1011
-123 = 1111 1000 0101
```

ขยาย"หนึ่ง" (ครื่องหมาย)

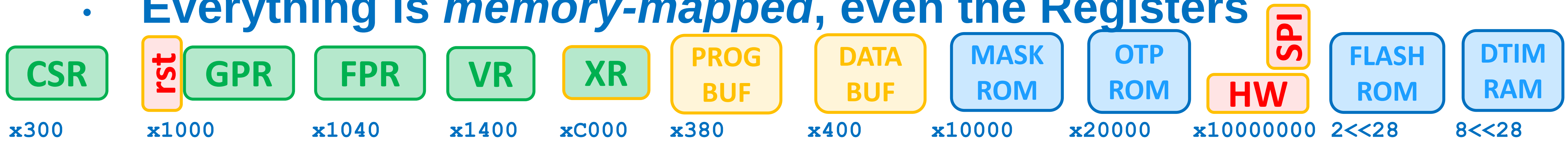
- Instructions like `sltiu` replace need for carry or borrow flags!

"Set if (the source register is) Less Than a sign-extended *I*mmediate, comparing as an Unsigned value

You will get used to `lui` ... `addi` as a common paradigm whenever you need to load a full 32-bit value

Variables Values

- ***Variables* are the Registers**
- **Instructions work with at most three Registers**
  Source #1 "`rs1`", Source #2 "`rs2`", and Destination "`rd`"
- **Everything is *memory-mapped*, even the Registers**



- **Can use the R-type instruction form to access a Register**

```
or s0, s0, s1  # x[s0] = x[s0] + x[s1]
```

- **Can use an I-type instruction form to Load or Store its value**

```
csrrs s0, 0x1008, s1  # x[s0] = x[s0] + x[s1]
```

s0 is a General Purpose Register also called "x8", thus, 8th register from x0, which starts at 0x1000

"Control and Status Register Read and Set contents of memory location 0x1008 with logical *OR* of S1"

Almost* All The Instructions You’ll Ever Need

- **Arithmetic Operations**
```
add    addi
sub
slt    slti
sltu   sltiu
```
- **Logical Operations**
```
and    andi
or     ori
xor    xori
```
- **(Big) Constant Operations**
```
auipc  lui
```
- **Unconditional Jumps**
```
jal    jalr
```
- **Bit-wise shifts**
```
sll    slli
srl    srli
sra    srai
```

37

- **Conditional Branches**
```
beq    bne
bge    bgeu
blt    bltu
```
- **Memory Loads and Stores**
```
lb     lbu     lh     lhu     lw
sb             sh             sw
```

*** You can “multiply” with repeated addition; “divide” with repeated subtraction; and do fractional “fixed point” arithmetic with left and right bit shifts.**

Quiz

- Do these have the same effect?
  What are the differences? What are the similarities?

```
or   x8, x8, x0
addi x8, x8, 0
slli x8, x8, 0
```

- Why is there no `subi` instruction? How else can you do this?

```
subi s0, s0, 5  # x[s0] = x[s0] - 5  ← ???
```

- How many ways can you mimic a `nop` instruction?

- Why is there no `ret` instruction? What's in its place?

- Why are these instructions *not* in the RISC-V ISA?

```
sla  slai  lwu  sbu  shu  swu
```

Extra Question: Where are the "Small Constant" operatons

The RISC-V Instruction Set

- **Grouped into “Extensions” designated by Aก, Bข, Cค, ..., Zห**
- **Mix or match as need and desire dictate or real Silicon allows**
- **Five most common instruction Extensions, in curious sequence**

  “I” Base – ***integer arithmetic & logical, constant values, jumps, memory load & store***
  “M” Mathematical – ***hardware multiply, divide, and remainder (modulus!)***
  “A” Atomic – ***read, modify, and write, in a single transaction with no interrupton***
  “C” Compressed – ***memory-saving 16-bit word sizes***
  “U” Privileged – ***Interrupts, Machine, Hypervisor, Supervisor, User modes***

Instruction Set is like a grammar or language

- **Other instruction Extensions are**

  “V” Vector operations, “B” Bit-manipulations, “H” Hypervisor privilege, “N” User mode int.
  “F D Q” Single-, Double-, Quad-precision FP, “J” Dynamic translations, “Z” Custom

AI
ML

Get involved with draft and ratification process!
https://github.com/riscv   https://riscv.org/technical/specifications/

How RISC-V Fits in the SoC

OpenOCD

USB, FT232R MPSSE, JTAG

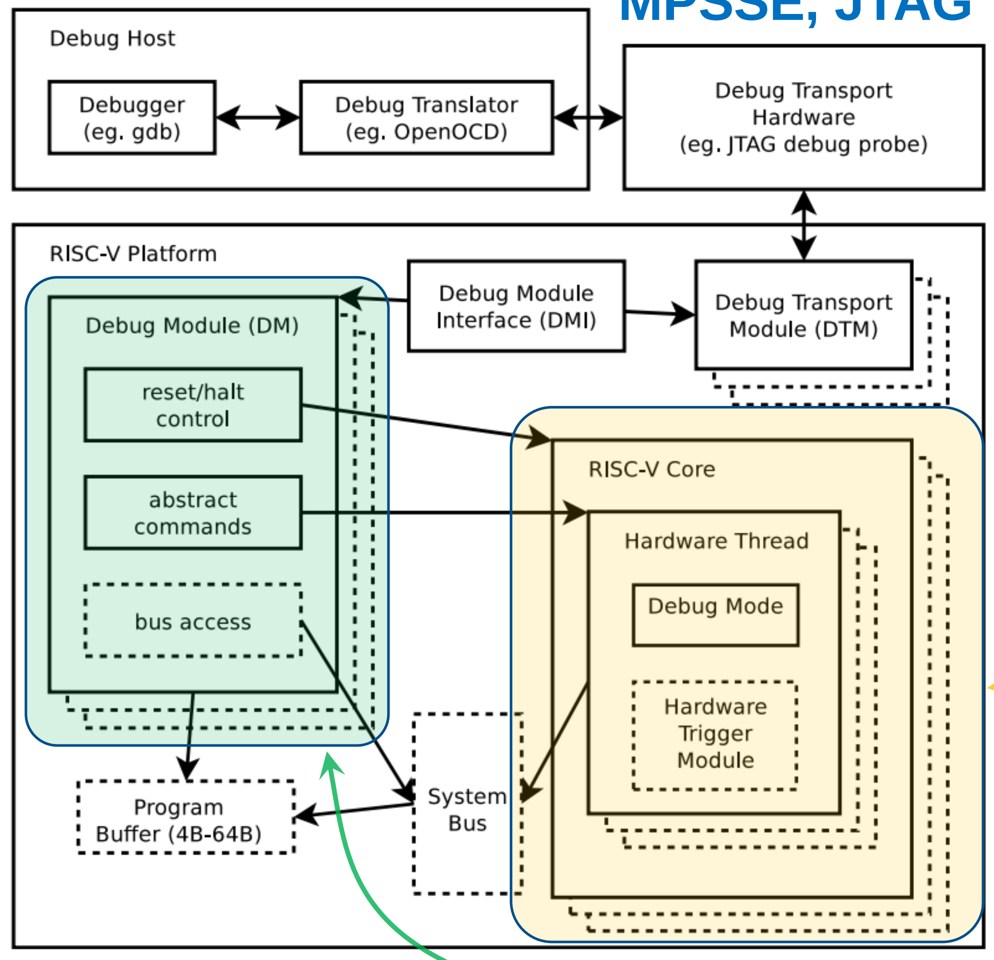


Figure 2.1: RISC-V Debug System Overview

Debug Specification – riscv-debug-release-0.13.2-20190522.pdf

FE310 -G002 SoC

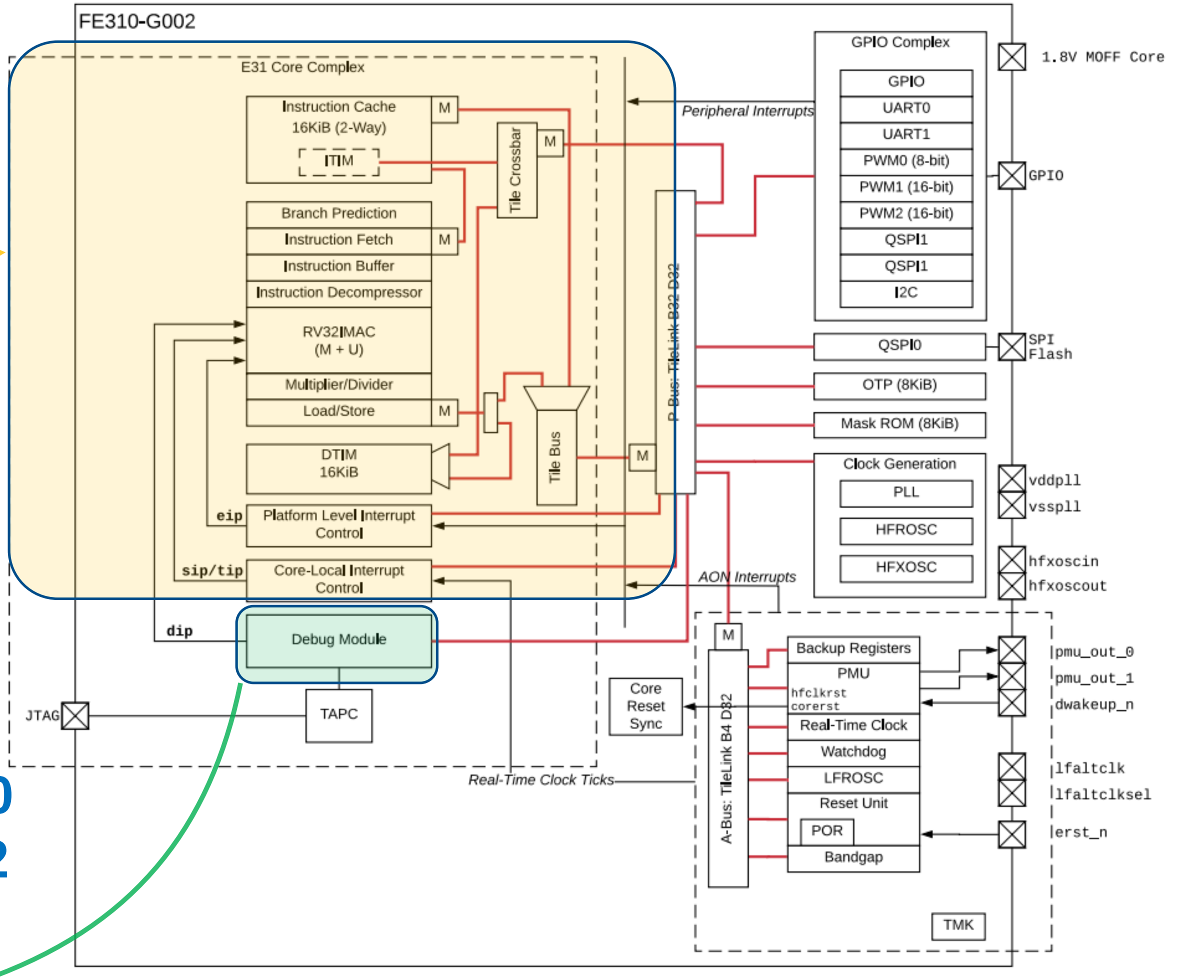


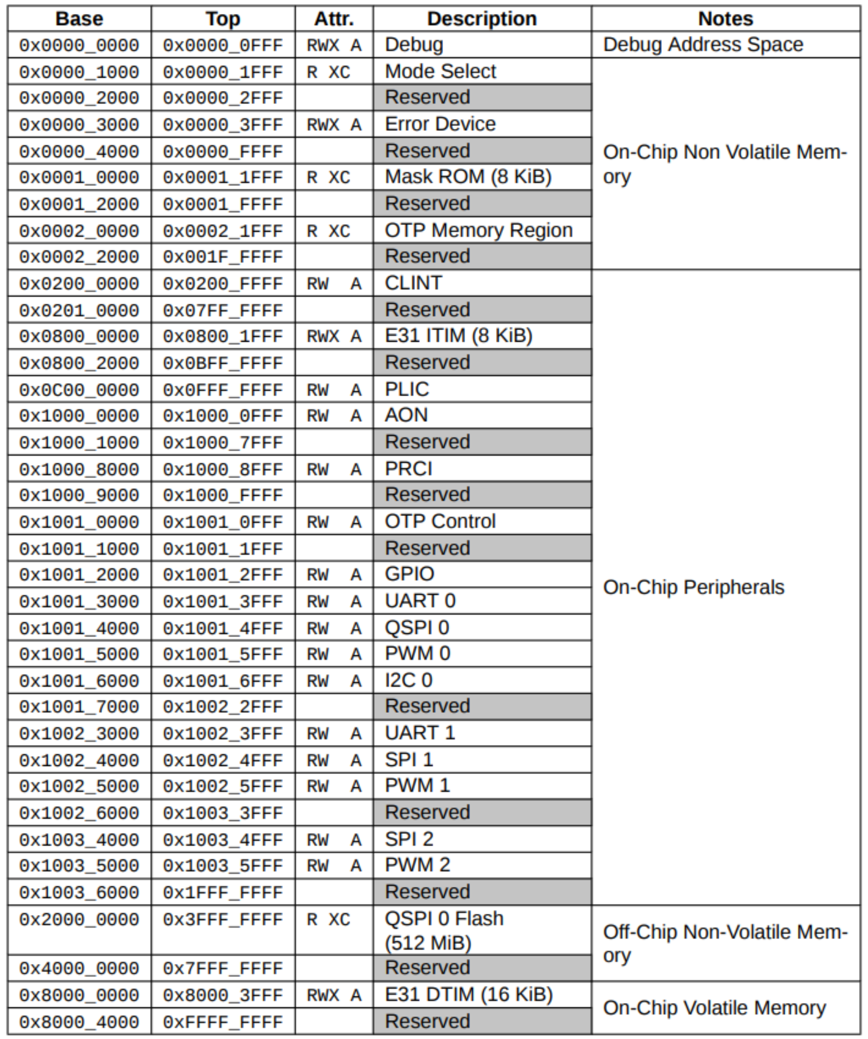
Figure 3.1: FE310-G002 top-level block diagram.

Unprivileged Architecture – riscv-spec-20191213.pdf

Privileged Architecture – riscv-privileged-20211203.pdf

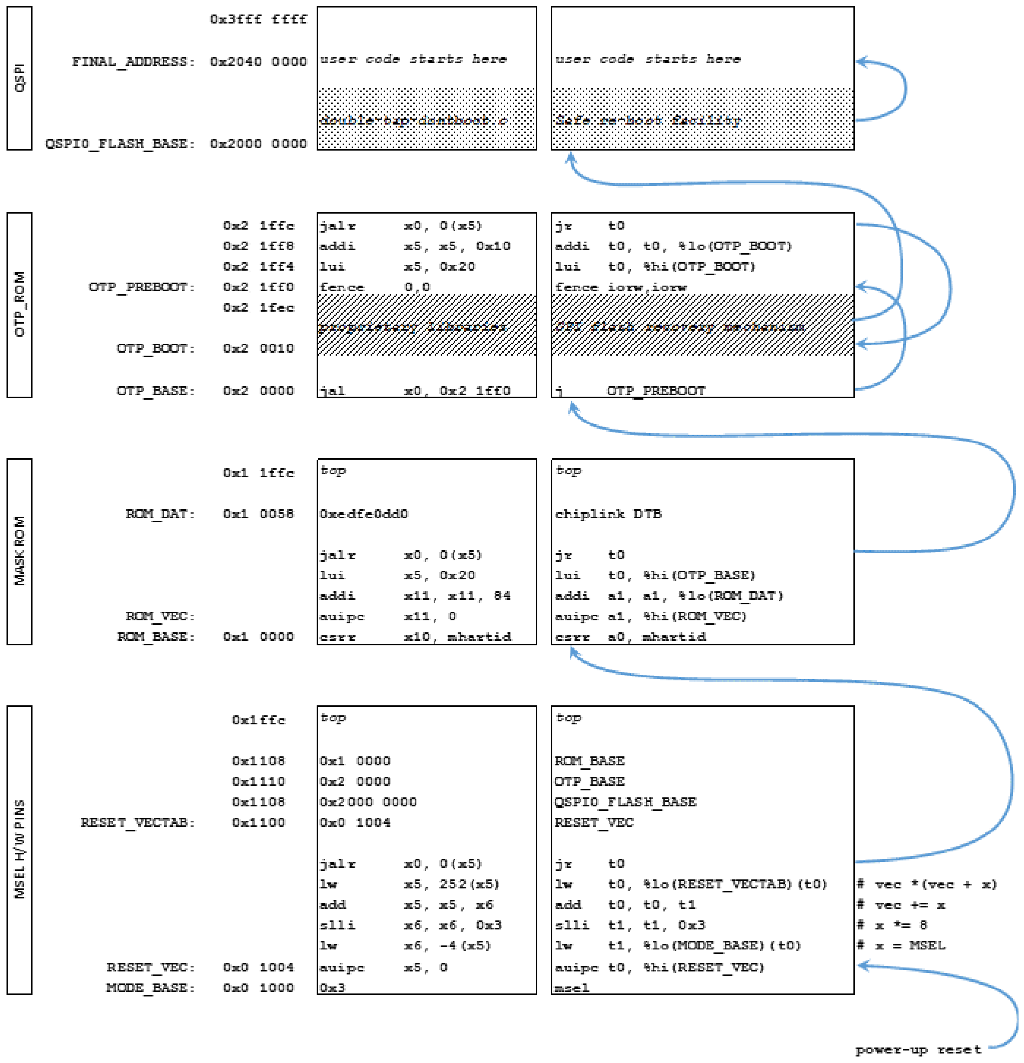
Register Map and Boot Flow

https://forums.sifive.com/t/is-there-c-boot-code-source-for-revb02-hardware/5987/2



Base	Top	Attr.	Description	Notes
0x0000_0000	0x0000_0FFF	RWX A	Debug	Debug Address Space
0x0000_1000	0x0000_1FFF	R XC	Mode Select	On-Chip Non Volatile Memory
0x0000_2000	0x0000_2FFF		Reserved	
0x0000_3000	0x0000_3FFF	RWX A	Error Device	
0x0000_4000	0x0000_FFFF		Reserved	
0x0001_0000	0x0001_1FFF	R XC	Mask ROM (8 KiB)	
0x0001_2000	0x0001_FFFF		Reserved	
0x0002_0000	0x0002_1FFF	R XC	OTP Memory Region	
0x0002_2000	0x001F_FFFF		Reserved	
0x0200_0000	0x0200_FFFF	RW A	CLINT	On-Chip Peripherals
0x0201_0000	0x07FF_FFFF		Reserved	
0x0800_0000	0x0800_1FFF	RWX A	E31 ITIM (8 KiB)	
0x0800_2000	0x0BFF_FFFF		Reserved	
0x0C00_0000	0x0FFF_FFFF	RW A	PLIC	
0x1000_0000	0x1000_0FFF	RW A	AON	
0x1000_1000	0x1000_7FFF		Reserved	
0x1000_8000	0x1000_8FFF	RW A	PRCI	
0x1000_9000	0x1000_FFFF		Reserved	
0x1001_0000	0x1001_0FFF	RW A	OTP Control	
0x1001_1000	0x1001_1FFF		Reserved	
0x1001_2000	0x1001_2FFF	RW A	GPIO	
0x1001_3000	0x1001_3FFF	RW A	UART 0	
0x1001_4000	0x1001_4FFF	RW A	QSPI 0	
0x1001_5000	0x1001_5FFF	RW A	PWM 0	
0x1001_6000	0x1001_6FFF	RW A	I2C 0	
0x1001_7000	0x1002_2FFF		Reserved	
0x1002_3000	0x1002_3FFF	RW A	UART 1	
0x1002_4000	0x1002_4FFF	RW A	SPI 1	
0x1002_5000	0x1002_5FFF	RW A	PWM 1	
0x1002_6000	0x1003_3FFF		Reserved	
0x1003_4000	0x1003_4FFF	RW A	SPI 2	
0x1003_5000	0x1003_5FFF	RW A	PWM 2	
0x1003_6000	0x1FFF_FFFF		Reserved	
0x2000_0000	0x3FFF_FFFF	R XC	QSPI 0 Flash (512 MiB)	Off-Chip Non-Volatile Memory
0x4000_0000	0x7FFF_FFFF		Reserved	
0x8000_0000	0x8000_3FFF	RWX A	E31 DTIM (16 KiB)	On-Chip Volatile Memory
0x8000_4000	0xFFFF_FFFF		Reserved	

Table 4: FE310-G002 Memory Map. Memory Attributes: **R** - Read, **W** - Write, **X** - Execute, **C** - Cacheable, **A** - Atomics

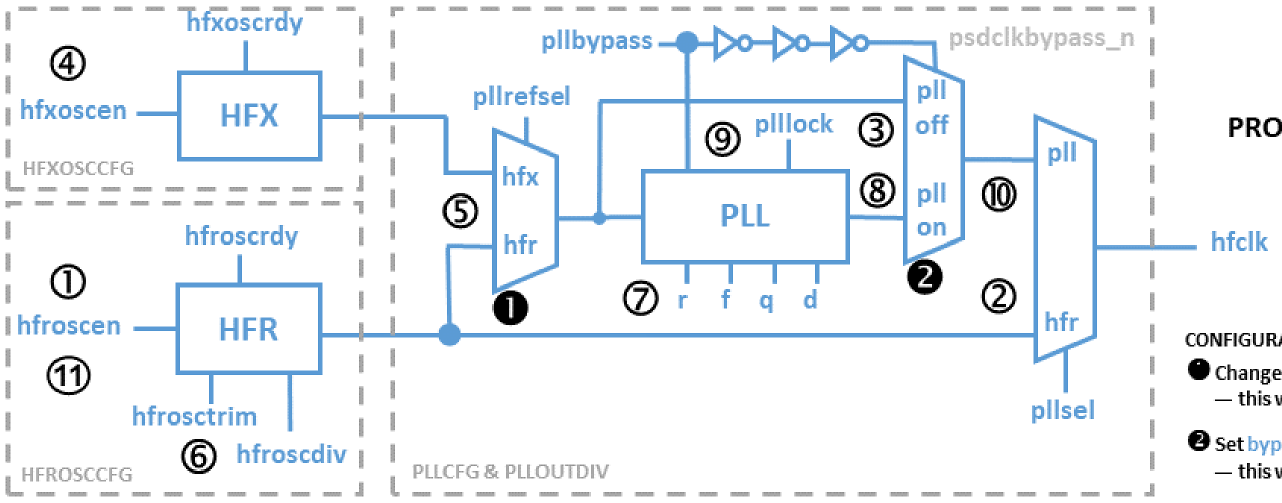


```
QSPI
                      0x3fff ffff
FINAL_ADDRESS:        0x2040 0000  user code starts here          user code starts here
QSPI0_FLASH_BASE:     0x2000 0000  double-tap-dontboot.c          Safe re-boot facility

OTP_ROM
                      0x2 1ffc     jalr    x0, 0(x5)              jr    t0
                      0x2 1ff8     addi    x5, x5, 0x10           addi  t0, t0, %lo(OTP_BOOT)
                      0x2 1ff4     lui     x5, 0x20               lui   t0, %hi(OTP_BOOT)
OTP_PREBOOT:          0x2 1ff0     fence   0,0                    fence iorw,iorw
                      0x2 1fec     proprietary libraries          OPT flash recovery mechanism
OTP_BOOT:             0x2 0010
OTP_BASE:             0x2 0000     jal     x0, 0x2 1ff0           j     OTP_PREBOOT

MASK ROM
                      0x1 1ffc     top                            top
ROM_DAT:              0x1 0058     0xedfe0dd0                     chiplink DTB
                                   jalr    x0, 0(x5)              jr    t0
                                   lui     x5, 0x20               lui   t0, %hi(OTP_BASE)
                                   addi    x11, x11, 84           addi  a1, a1, %lo(ROM_DAT)
ROM_VEC:                           auipc   x11, 0                 auipc a1, %hi(ROM_VEC)
ROM_BASE:             0x1 0000     csrr    x10, mhartid           csrr  a0, mhartid

MSEL H/W PINS
                      0x1ffc       top                            top
                      0x1108       0x1 0000                       ROM_BASE
                      0x1110       0x2 0000                       OTP_BASE
                      0x1108       0x2000 0000                    QSPI0_FLASH_BASE
RESET_VECTAB:         0x1100       0x0 1004                       RESET_VEC
                                   jalr    x0, 0(x5)              jr    t0
                                   lw      x5, 252(x5)            lw    t0, %lo(RESET_VECTAB)(t0)  # vec *(vec + x)
                                   add     x5, x5, x6             add   t0, t0, t1                 # vec += x
                                   slli    x6, x6, 0x3            slli  t1, t1, 0x3                # x *= 8
                                   lw      x6, -4(x5)             lw    t1, %lo(MODE_BASE)(t0)     # x = MSEL
RESET_VEC:            0x0 1004     auipc   x5, 0                  auipc t0, %hi(RESET_VEC)
MODE_BASE:            0x0 1000     0x3                            msel
```

After reset, flow flow ends at base of Flash (1<<17)

Using Debug, can halt and reume anywhere, like base of RAM (1<<31)

Clock Path and Configuration Procedure



FE310 PRCI
SYSTEM CLOCK
PROGRAMMING SEQUENCE STEPS

not every step need by used when changing from one system clock configuration to another

CONFIGURATION RULES:

❶ Change refsel only when sel=hfr and bypass=pll off
— this will prevent possible hang-up and pll unlock

❷ Set bypass=pll on only when sel=hfr
— this will allow pll settle and give time to check lock

rule	*step*	*description*	*signal(s) used*
	1	Enable HFR OSC, and wait for rdy signal	HFROSCCFG hfroscen[30] = 1, hfroscrdy[31] == 1
	2	Select intermediate clock path, bypassing everything	PLLCFG pllsel[16] = 0
	3	Power off PLL, and bypass around it	PLLCFG pllbypass[18] = 0
	4	Enable or disable HFX OSC, as desired, and wait for rdy	HFXOSCCFG hfxoscen[30] = 1 | 0, hfxoscrdy[31] == 1
❶	5	Select clock source reference, HFX or HFR, as desired	PLLCFG pllrefsel[17] = 1 | 0
	6	Configure HFR OSC trim and divisor parameters	HFROSCCFG hfrosctrim[20:16], hfroscdiv[5:0]
	7	Configure PLL multiplier (f) and divisor (r, q, d) parameters	PLLCFG r[2:0], f[9:4], q[11:10]; PLLOUTDIV d[5:0]
❷	8	Power on PLL, and select through it	PLLCFG pllbypass[18] = 1
	9	Wait for PLL lock signal	PLLCFG plllock[31] == 1
	10	Restore normal clock path	PLLCFG pllsel[16] = 1
	11	Disable HFR OSC	HFROSCCFG hfroscen[30] = 0

https://github.com/psherman42/Demonstrating-PRCI

A Few Easy Steps to get Up and Running

- **Install assembler, compiler, and linker**
  riscv64-unknown-elf-toolchain-10.2.0-2020.12.8-x86_64-w64-mingw32.zip
- **Install loader**
  xpack-openocd-0.11.0-5-win32-x64.zip
- **Set environment variables – point to '`/bin`' folders**
- **Install Gnu `make.exe` program – anywhere in path**
- **Use Zadig utility to replace ftdi drivers with `libusbK`**
- **Configure your `jcsse.mk` file**
- **Build your project:** `make -f jcsse.mk ramload`
  or one of: `ramrun | ramdebug | romload | romrun | romdebug`

Anatomy of the Project

- **“Define your program here” in the `jcsse.mk` file**
- **`start.s` – 1st thing that happens at reset or power-up**
  because _start gets “`-ld`” linked to start of RAM/ROM
- **Follow the “`jal`”’s from `s_main` to `c_main`**
- **Use Notepad, TextPad, vi(m), etc., for best results**

Elements of Coding Style

C

Header File .h .e

```
//----- cfile.h -----------
#ifndef CFILE_H__INCLUDED
#define CFILE_H__INCLUDED

#define SOME_C_THING 0x12345

#endif//CFILE_H__INCLUDED
```

The "Guarded Include"

Source File .c .s

```
//------- cfile.c --------
#include "cfile.h"

uint32_t c_func (uint32_t n)
{
  return n += 42;  // everything
}
```

Assembly

```
#------ afile.e -----------
.ifndef AFILE_E__INCLUDED
.equ AFILE_E__INCLUDED,1

.equ SOME_A_THING, 0x12345

.endif  # AFILE_E__INCLUDED
```

```
#-------- afile.s ---------
.include «afile.e»

a_func: .global a_func  # public
  addi a0, a0, 42  # everything
  ret
```

The GNU Make Toolchain .mk

1. Compile
2. Assemble

```
start.o : start.s
	$(RVGNU)-as $(AOPS) start.s -o start.o
gpio.o : gpio.s
	$(RVGNU)-as $(AOPS) gpio.s -o gpio.o
$(PROGRAM).o : $(PROGRAM).c
	$(RVGNU)-gcc $(COPS) -c $(PROGRAM).c -o $(PROGRAM).o
```

```
PROGRAM := main
OBJECTS = start.o \
          $(PROGRAM).o \
           gpio.o
```

3. Link

```
$(PROGRAM).elf : my-soc.lds $(OBJECTS)
	$(RVGNU)-ld $(OBJECTS) -g -T my-soc.lds -o $(PROGRAM).elf -Map
$(PROGRAM).map $(LKOPS)
	$(RVGNU)-objdump -D $(PROGRAM).elf > $(PROGRAM).lst
```

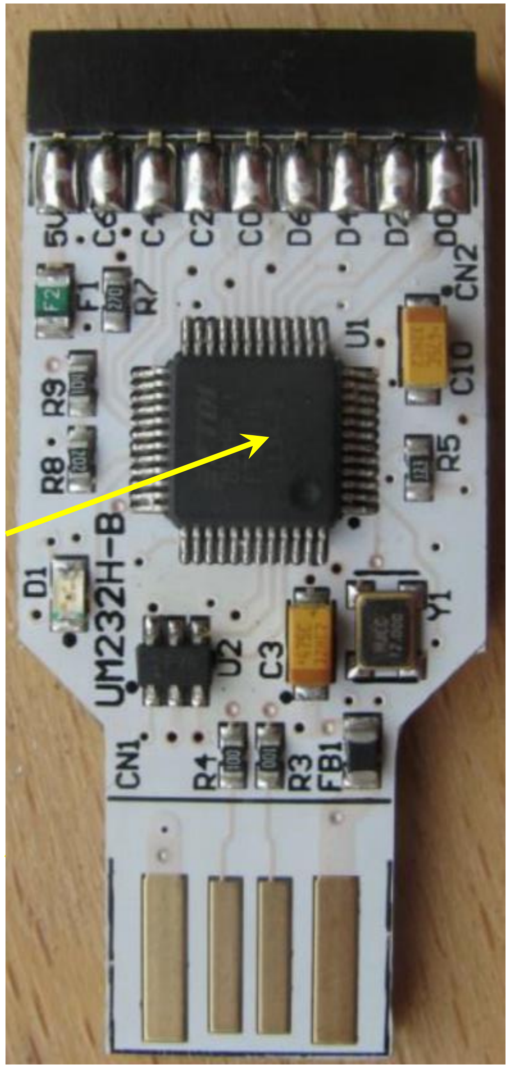
4. Load

```
$(PROGRAM)-ram.bin : $(PROGRAM).elf
	$(RVGNU)-objcopy --dump-section .text=$(PROGRAM)-ram.bin $(PROGRAM).elf
ramload : $(PROGRAM)-ram.bin
	@openocd -f interface.cfg -f my-soc.cfg -c init -c "reset init" -c
"asic_ram_load $(PROGRAM) 0x80000000 no_run" -c shutdown -c exit
```

Indented by tabs `\t` not spaces!

The Hardware

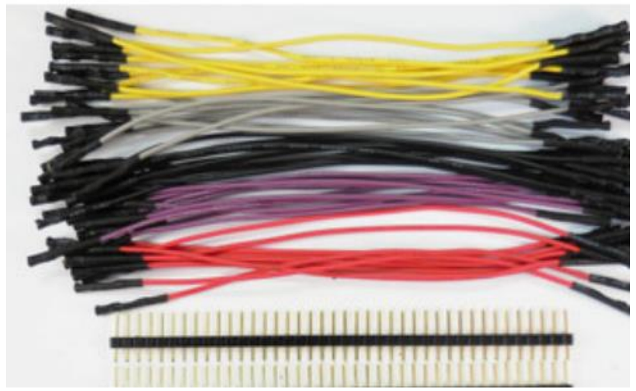
UM232H-B



FTDI
FT232H
(MPSSE)

5V, 500 mA
USB-A

920-0192-50

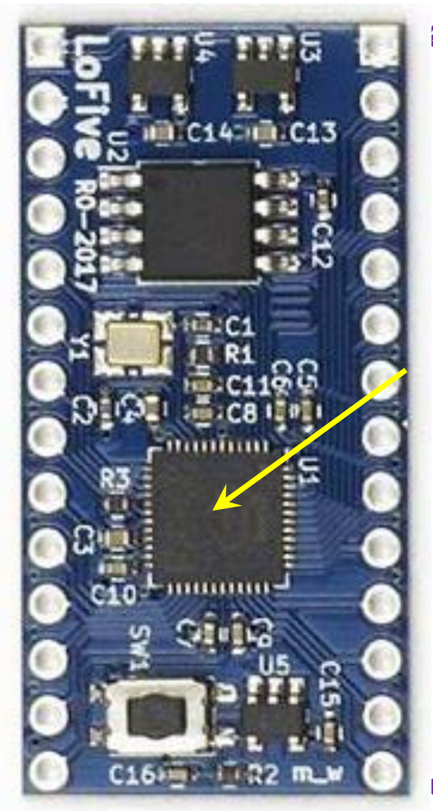


TC2030-FTDI-C232HD-EDHSP-0

LOFIVE-R1

Pin 1

Pin 28



SiFive
FE310-G002
SoC

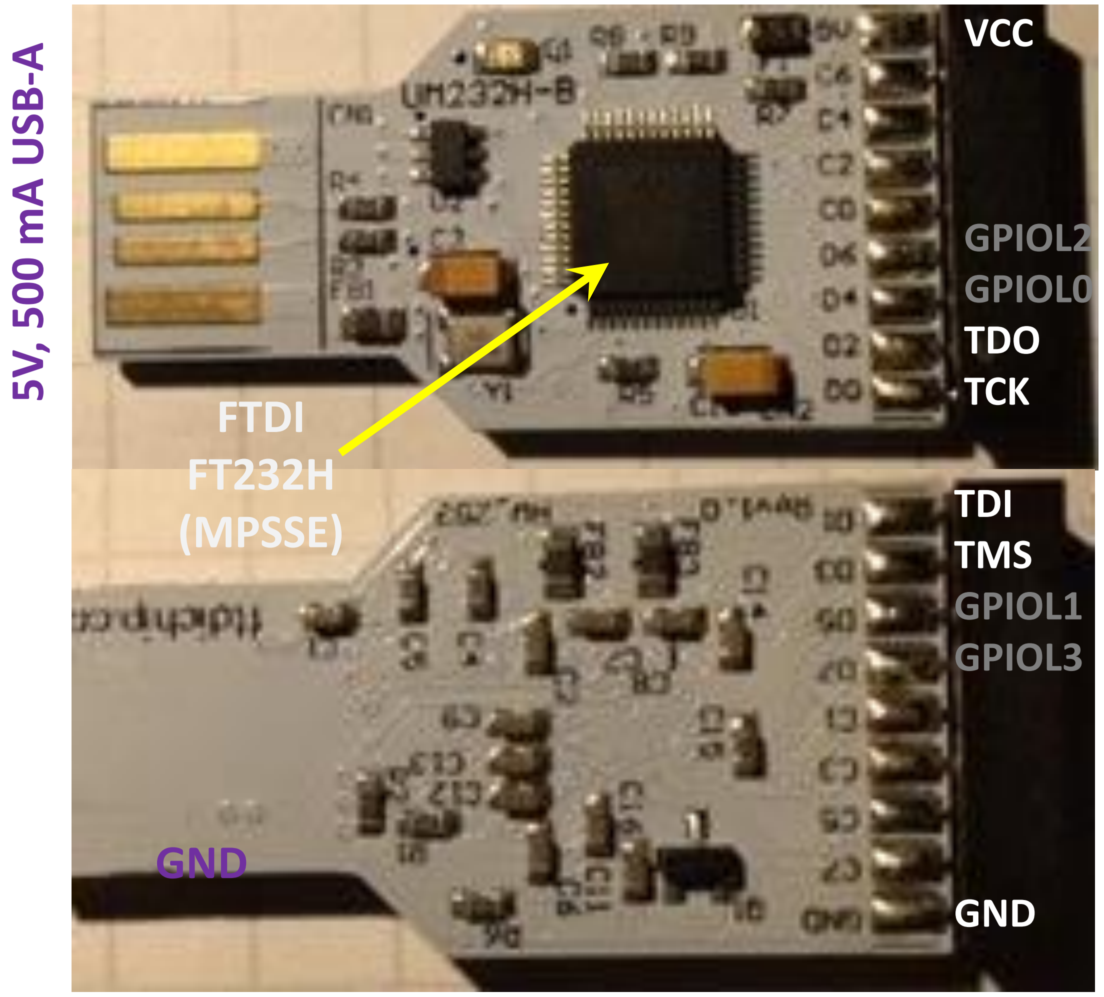
Pin 14

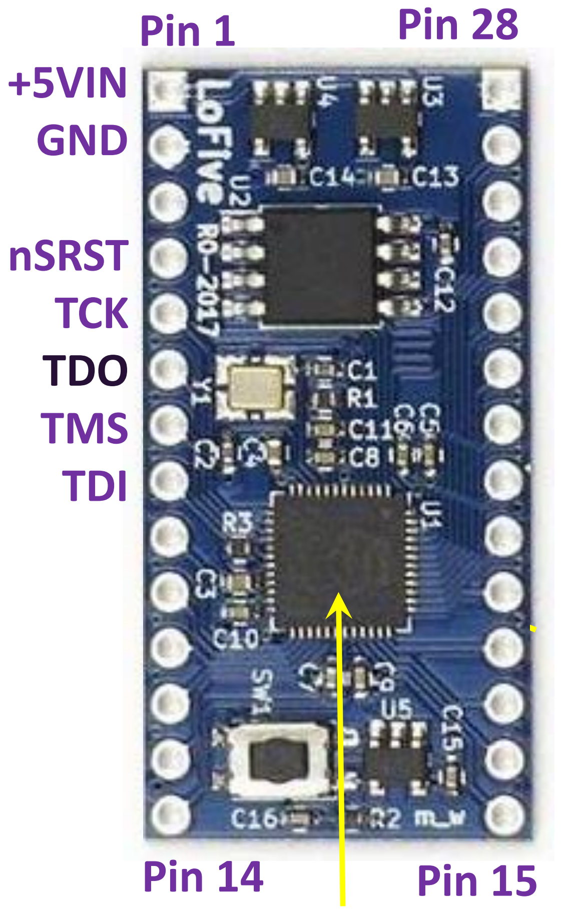
Pin 15

FTDI JTAG Cable Hardware

UM232H-B

LOFIVE-R1





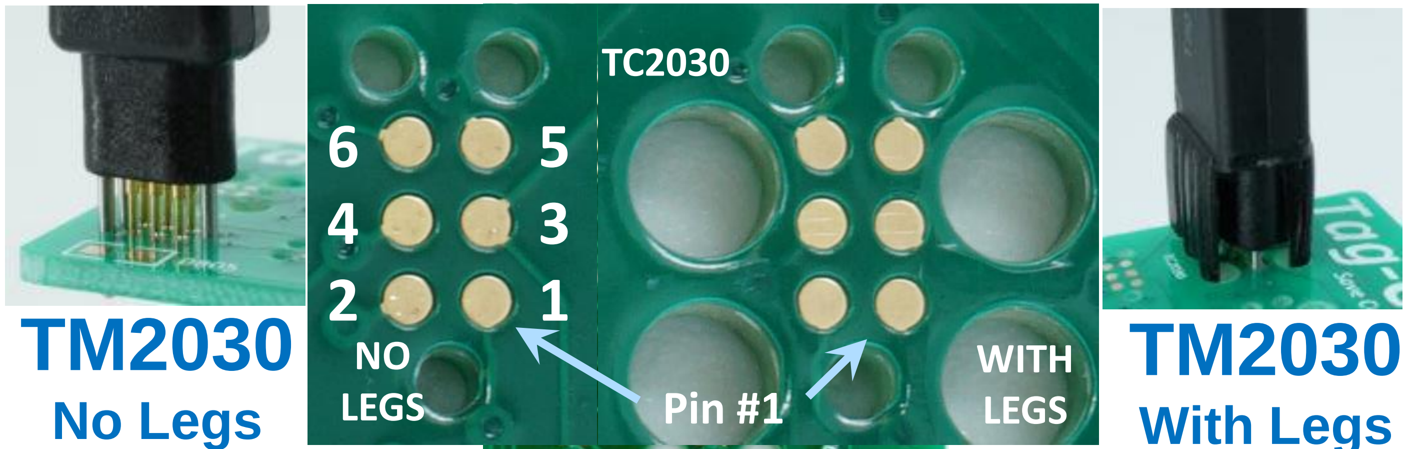
SiFive
FE310-G002 SoC

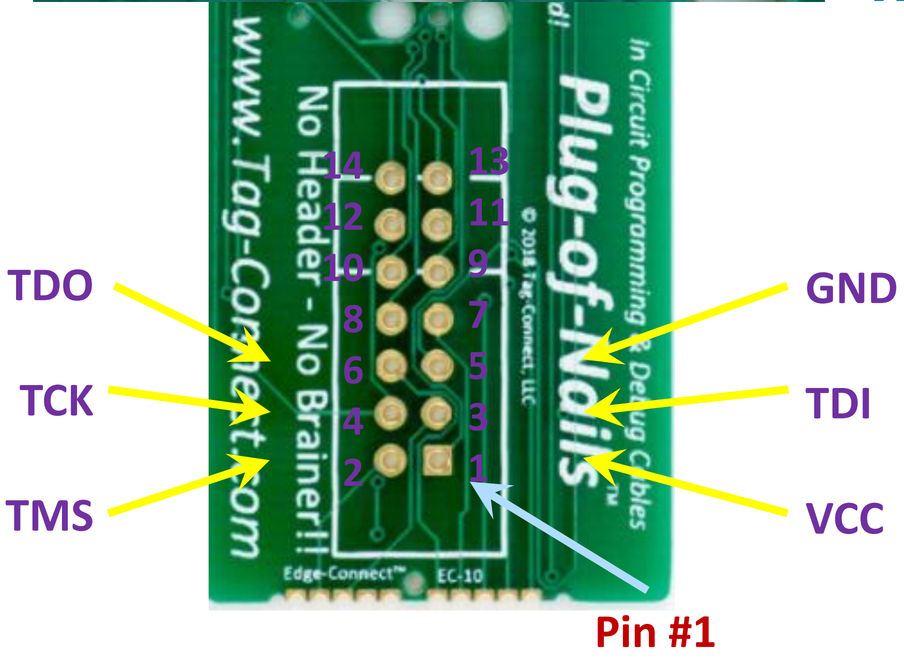
Vcc = 5 Vdc
Icc <250 mA
Vio 3.3 Vdc
D.R. <12 Mbps

Tag-Connect JTAG Cable Hardware

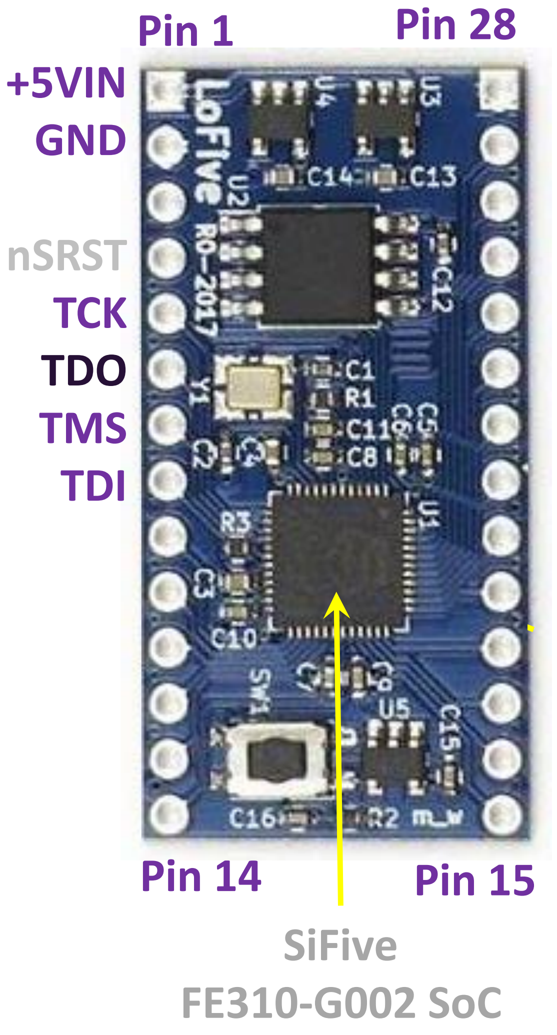
TC2030-NL-FTDI-C232HD-DDHSP-0

TC2030-FTDI-C232HD-DDHSP-0





LOFIVE-R1



Vcc 3.3 Vdc
Icc <250 mA
Vio 3.3 Vdc
D.R. <12 Mbps

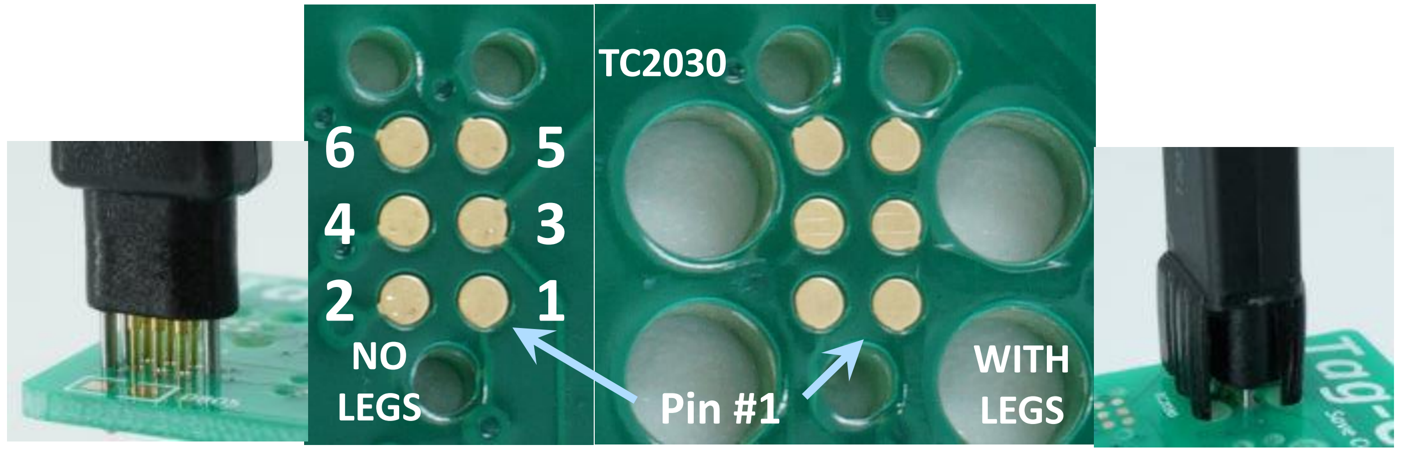
Tag-Connect Pin Numbering Style

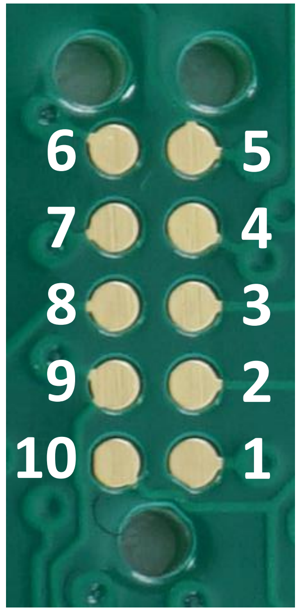
TM2030
No Legs

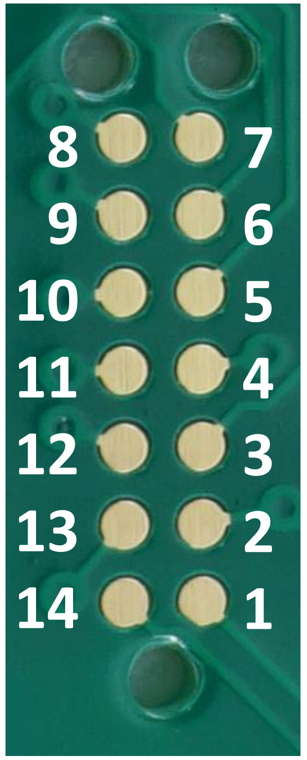
TM2030
With Legs

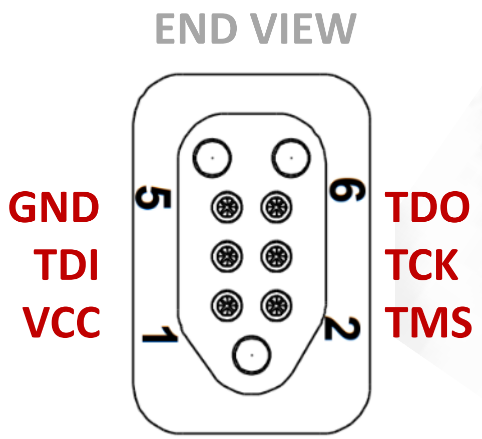
TM2050
No Legs

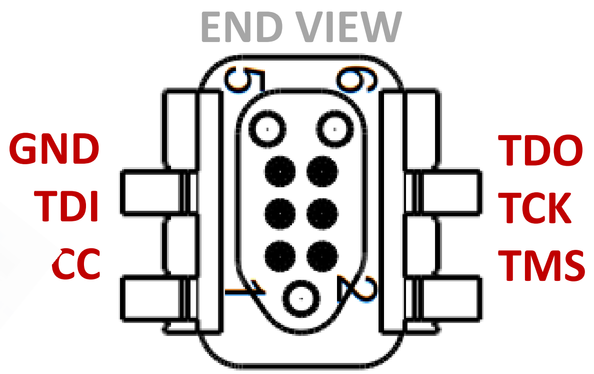
TM2070
No Legs











Tag-Connect JTAG Cable Reference Info

Mechanical Footprint Library https://www.tag-connect.com/info/footprint-downloads

How To Use https://www.tag-connect.com/technical/using-tag-connect

FTDI Selection Guide https://www.tag-connect.com/product-category/products/cables/ftdi

TM2030
No Legs

Mechanical Outline and PCBA Layout

https://www.tag-connect.com/wp-content/uploads/bsk-pdf-manager/2019/12/TC2030-IDC-NL-Datasheet-Rev-B.pdf

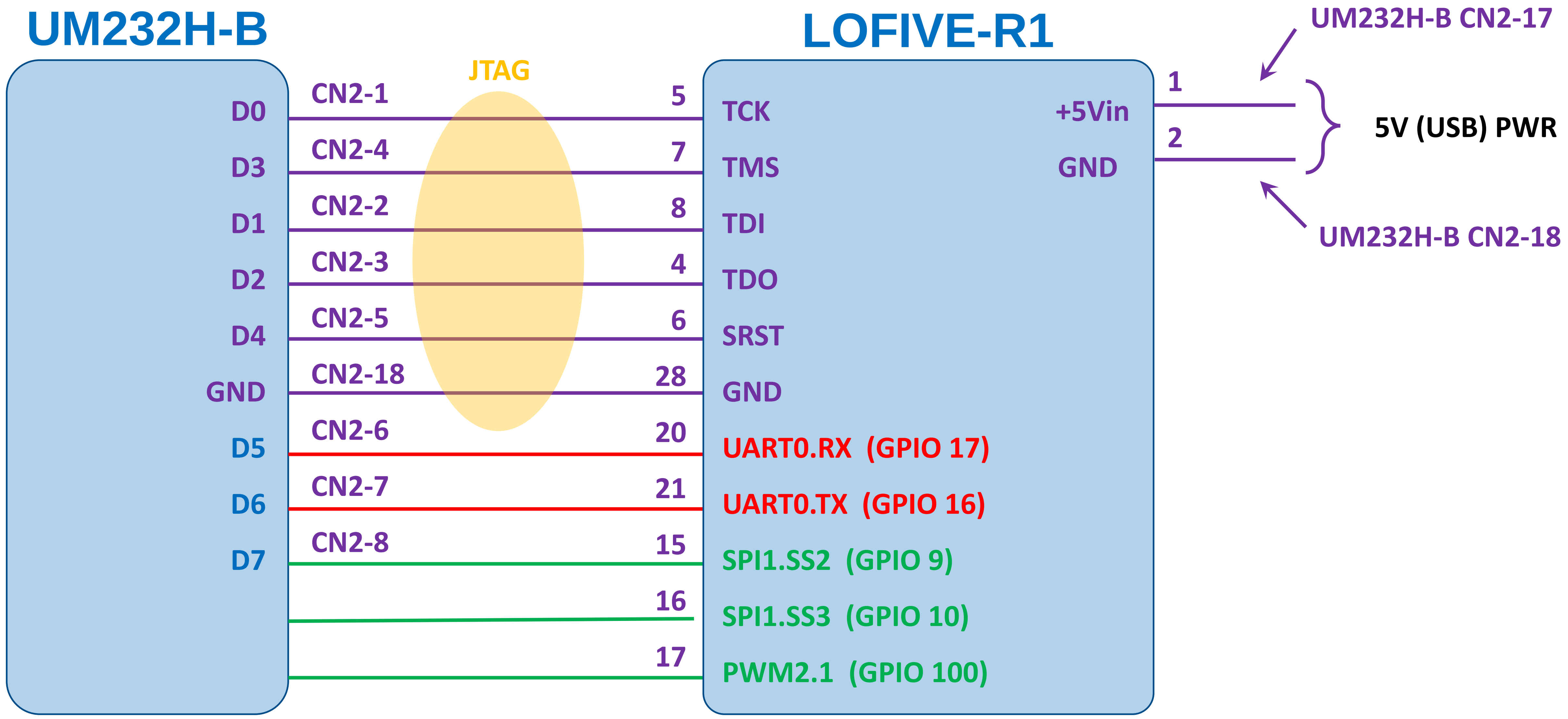
https://www.tag-connect.com/wp-content/uploads/bsk-pdf-manager/2019/12/TC2030-IDC-Datasheet-Rev-B.pdf

TM2030
With Legs

Data Sheet TC2030-FTDI-C232HD-DDHSP-0
https://www.tag-connect.com/product/tc2030-ftdi-c232hd-ddhsp-0

Data Sheet TC2030-NL-FTDI-C232HD-DDHSP-0
https://www.tag-connect.com/product/tc2030-nl-ftdi-c232hd-ddhsp-0

Wiring the Hardware



Documentation and Reference Material

Unprivileged Architecture – riscv-spec-20191213.pdf

Privileged Architecture – riscv-privileged-20211203.pdf

https://www.riscv.org/technical/specifications

Debug Specification – riscv-debug-release-0.13.2-20190522.pdf

https://github.com/riscv/riscv-debug-spec

"B" Extension – riscv-bitmanip-0.92-20191108.pdf

E31 Core Complex – SiFive-E31-Manual-v2p0.pdf

https://static.dev.sifive.com/SiFive-E31-Manual-v2p0.pdf

FE310-G002 Manual – fe310-g002-v1p5.pdf

https://sifive.cdn.prismic.io/sifive/034760b5-ac6a-4b1c-911c-f4148bb2c4a5_fe310-g002-v1p5.pdf

FE310-G002 Datasheet – fe310-g002-manual-v19p05.pdf

https://sifive.cdn.prismic.io/sifive/4999db8a-432f-45e4-bab2-57007eed0a43_fe310-g002-datasheet-v1p2.pdf

FT232H Datasheet – DS_FT232H.pdf

http://www.ftdichip.com/Support/Documents/DataSheets/Modules/DS_UM232H-B.pdf

---

LoFive Schematic Diagram – lofive-r1.pdf

https://github.com/mwelling/lofive

USB JTAG Module Datasheet – DS_UM232H-B.pdf

http://www.ftdichip.com/Support/Documents/DataSheets/ICs/DS_FT232H.pdf

Cheat Sheets & Further Reading

RISC-V Reference "Green" Card – riscv-greencard-20181213.pdf

http://riscbook.com/greencard-20181213.pdf

GNU as Assembler Directives – GNU_Assembler_Directives.pdf

GNU gdb Debugger – gdb-quick-reference-card.pdf

User: paul
Pwd: paul

The RISC-V Reader: An Open-Architecture Atlas

http://192.142.160.4/~sherman/naruemon/riscbook/risc-v-reader-thai-v2p0d4.pdf

คู่มือ RISC-V: สถาปัตยกรรมแบบเปิด – risc-v-reader-thai-v2p0d4.pdf

http://riscbook.com/

Interrupt Cookbook – sifive-interrupt-cookbook-v1p2.pdf

https://www.sifive.com/documentation

Good Discussions – https://forums.sifive.com

https://forums.sifive.com
See HiFive1 Rev B, user: **pds**

Entire Scala/Chisel Chip Design – sifive-blocks

https://github.com/sifive/sifive-blocks

Other Hardware In The Landscape

RPi – Pinout Diagram

Olimex Adapters – ARM-USB-TINY-H

FTDI Modules – ARM-USB-TINY-H

Interconnects – Tag-Connect

https://pinout.xyz

olimex

https://ftdichip.com/product-category/products/modules/

tag-connect

Available at distributors like: digikey, mouser, adafruit

Now To The Lab

♪ ไปสนุกกัน ♬
ธีร์ ไชยเดช

- Explore an Extension of your choice from RISC-V ISA alphabet
- Use marker code and oscilloscope to see how your code works

```
some_code:
  #-------- marker pulse init --------
  li t5, (1<<21)
  lui t6, 0x10012     # GPIO_BASE
  addi t4, t6, 0x08   # GPIO_BASE + GPIO_OUTPUT_EN
  amoor.w x0, t5, (t4)  # gpio_output_en |= (1<<21)
  addi t6, t6, 0x0C   # GPIO_BASE + GPIO_OUTPUT_VAL
#-------- marker pulse init --------
  li t3, 5000
some_code_loop:
  # ...
  amoxor.w x0, t5, (t6)  # MARKER PULSE: gpio_output_val ^= (1<<21)
  # ...
  addi t3, t3, -1
  bgez t3, some_code_loop
  #
  ret
```

Initialize the marker signal
In this case, GPIO21 is
pin #4R on the LoFive board

Copy/paste wherever you like.

May also use separate `lw` and `sw` style, too – what does it look like?

Select Your Team

	code file	*The RISC-V Reader*
The Atomics – RV32A	*`atom.s`*	*Ch. 6*
The Privileged – RV32U	*`priv.s`*	*Ch. 10*
The Compressed – RV32C	*`tiny.s`*	*Ch. 7*
The Mathematicians – RV32M	*`math.s`*	*Ch. 4*
The Builders of the Base – RV32I	*`base.s`*	*Ch. 2*
The Assemblers – Toolchain	*`assm.s`*	*Ch. 3*

user: **paul**
pwd: **paul**

http://192.142.160.4/~sherman/naruemon/riscbook

The Atomics (RV32A) *Ch. 6*

- **Load – modify – store – or, Read-Modify-Write**
- **`Amoxxx` and `lr` / `sc`**
- **Memory alignment**
- **`x[rd]` before the mod**

```
t ← M[rs1]
M[rs1] ← t op x[rs2]
x[rd] ← t
```

a hardware FIFO like a TX buffer that's FULL might later succeed in its operating, storing ...

... yet the AMOxxxx result might erroneously show "failure"

The Privileged (RV32U) *Ch. 10*

- **User, Supervisor, Hypervisor, Machine modes**
- **Asynchronous Interrupts and Synchronous Exceptions**
- **Vector tables and alignments**
- **Enabling and clearing pending bits**
- **CLINT & PLIC**

The Compressed (RV32C) *Ch. 7*

- **Half-size but fewer registers**
- **How to force, how to prevent use of**

The Mathematicians (RV32M) *Ch. 4*

- **Hardware multiply and divide**
- **`Rem(u)` instruction as modulus**

The Builders Of The Base (RV32I) *Ch. 2*

- Arithmetic and logic instructions
- Carry and Borrow
- Immediate and register instruction forms
- Long-immediate and Short-immediate constants
- `lui` and `auipc`
- The Wonderful "0"

  "mv" is just an add ... j and jr are just jal and jalr
- Conditional and Unconditional Jump

The Assemblers (Toolchain) *Ch. 3*

- **Compile, assemble, link, load, debug**
- **How can you improve the `jcsse.mk` file?**
- **`gnu`, `git`, and (Xe)(La)(TeX)**
- **How and where to get all for free**
- **How to build all tool programs from scratch**

Acknowledgments

- **Sarin Termsuta – as first point-of-contact, for believing in me**
- **Mike, Modem, Tong, all NU folk – for chosing a Great place to study**
- **Dr. Wansuree and Dr. Kraisak – seeing light at end of tunnel**

Contacts

Paul Sherman
พอลเชอร์แมน

pauldylansherman@icloud.com

+1 (510) 229-6249

Github: psherman42

LINE: pauldsherman

linkedin.com/in/psherman

Naruemon Ratanakunakorn
นฤมล

naruemon323059@gmail.com

+66 (0) 6-5087-2687