

Admin

Assign0 (git workflow)

due before Tue lab

Lab I Tue & Wed

meet your Mango Pi!

Assign I released after Wed lab

Larson scanner



Today: Let there be light!

More on RISC-V assembly, instruction encoding

Peripheral access through memory-mapped registers

Goal: blink an LED

Challenge for you all:

Write an assembly program to count the "on" bits in a given numeric value

```
li a0, some-number  
li a1, 0
```

```
// a0 initialized to input value  
// use a1 to store count of "on" bits in value
```



Ripes visual simulator

The screenshot shows the Ripes visual simulator interface. On the left, there's a sidebar with icons for Processor, Cache, Memory, and I/O, and a stack of four hex values: 100, 1010, 01, and Editor. The main area has tabs for Source code, Input type (Assembly selected), Executable code, View mode (Disassembled selected), and a search/filter icon. The source code editor contains the following assembly code:

```
1 li a0, 67
2 li a1, 0
3 loop:
4     andi a2,a0,1
5     add a1,a1,a2
6     srli a0,a0,1
7     bne a0,x0,loop
8
```

The disassembly view shows the following instructions:

Address	OpCode	Description
0:	04300513	addi x10 x0 67
4:	00000593	addi x11 x0 0
8:	00157613	andi x12 x10 1
c:	00c585b3	add x11 x11 x12
10:	00155513	srli x10 x10 1
14:	fe051ae3	bne x10 x0 -12 <loop>

On the right, there's a register window titled 'gpr' with a table:

Name	Alias	Value
x0	zero	0x00000000
x1	ra	0x00000000
x2	sp	0x7fffffff
x3	gp	0x10000000
x4	tp	0x00000000
x5	t0	0x00000000
x6	t1	0x00000000
x7	t2	0x00000000
x8	s0	0x00000000
x9	s1	0x00000000
x10	a0	0x00000000
x11	a1	0x00000003
x12	a2	0xaaaaaaaaaa1

Try it yourself! <https://ripes.me>

Instruction encoding

Another way to understand the design of an ISA is to look at how the bits are used in the instruction encoding.

In RISC-V, each instruction is encoded in **32*** bits.

Instructions are organized into six groups, some features are common within/across groups.

The encoding design intentionally reuses same bits for common features to simplify decode circuitry.

*Compressed extension adds 16-bit compact encoding for some insns

RISC-V instruction encoding

32-bit RISC-V instruction formats																																			
Format	Bit																																		
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
Register/register	funct7							rs2					rs1					funct3			rd			opcode											
Immediate	imm[11:0]							rs1					funct3			rd			opcode																
Store	imm[11:5]							rs2					rs1			funct3			imm[4:0]					opcode											
Branch	[12]	imm[10:5]						rs2					rs1			funct3			imm[4:1]				[11]	opcode											
Upper immediate	imm[31:12]																																		
Jump	[20]	imm[10:1]						[11]	imm[19:12]							rd			rd			opcode													
<ul style="list-style-type: none"> • opcode (7 bits): Partially specifies one of the 6 types of <i>instruction formats</i>. • funct7 (7 bits) and funct3 (3 bits): These two fields extend the <i>opcode</i> field to specify the operation to be performed. • rs1 (5 bits) and rs2 (5 bits): Specify, by index, the first and second operand registers respectively (i.e., source registers). • rd (5 bits): Specifies, by index, the destination register to which the computation result will be directed. 																																			

6 instruction types

Regularity in bit placement to ease decoding

Sparse instruction encoding (room for growth)

ALU R-type encoding

31	27	26	25	24	20	19	15	14	12	11	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 10:5]		rs2		rs1		funct3		imm[4:1 11]		opcode			B-type	
		imm[31:12]						rd		opcode			U-type	

add **x3, x1, x2**

0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

00000000**00010000010000001000001000001000110110011**

0 0 2 0 8 1 B 3

ALU I-type encoding

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
funct7		rs2			rs1		funct3		rd		opcode		R-type	
imm[11:0]					rs1		funct3		rd		opcode		I-type	

imm[11:0]	rs1	000	rd	0010011	ADDI
imm[11:0]	rs1	010	rd	0010011	SLTI
imm[11:0]	rs1	011	rd	0010011	SLTIU
imm[11:0]	rs1	100	rd	0010011	XORI
imm[11:0]	rs1	110	rd	0010011	ORI
imm[11:0]	rs1	111	rd	0010011	ANDI

Your turn!

addi a0, zero, 21

00000001010100000000010100010011
 0 1 5 0 0 5 1 3

Know your tools: assembler

The *assembler* reads assembly instructions (*text*) and outputs as machine-code (*binary*). The reverse process is called *disassembly*.

This translation is mechanical, fully deterministic.

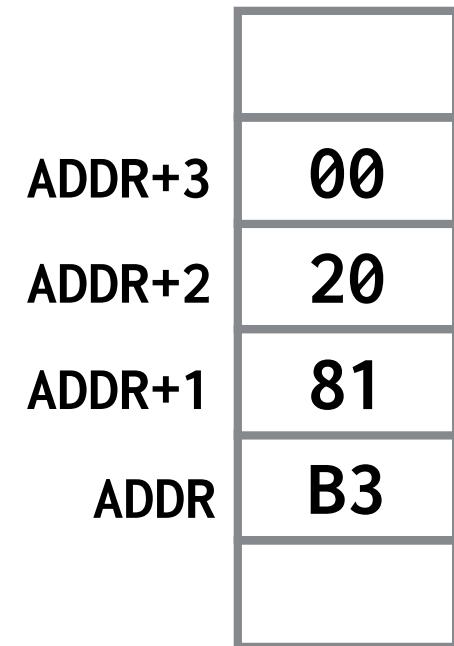
```
$ riscv64-unknown-elf-as add.s -o add.o  
  
$ ls -l add.o  
928 add.o  
  
$ riscv64-unknown-elf-objcopy add.o add.bin -O binary  
  
$ ls -l add.bin  
4 add.bin  
  
$ hexdump -C add.bin  
00000000  b3 81 20 00
```

MangoPi memory is little-endian

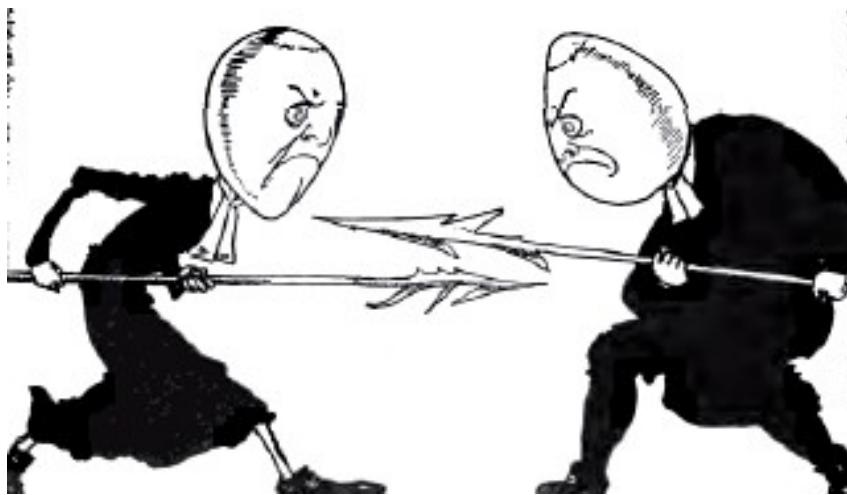
most-significant-byte (MSB)



least-significant-byte (LSB)



little-endian
(LSB first)

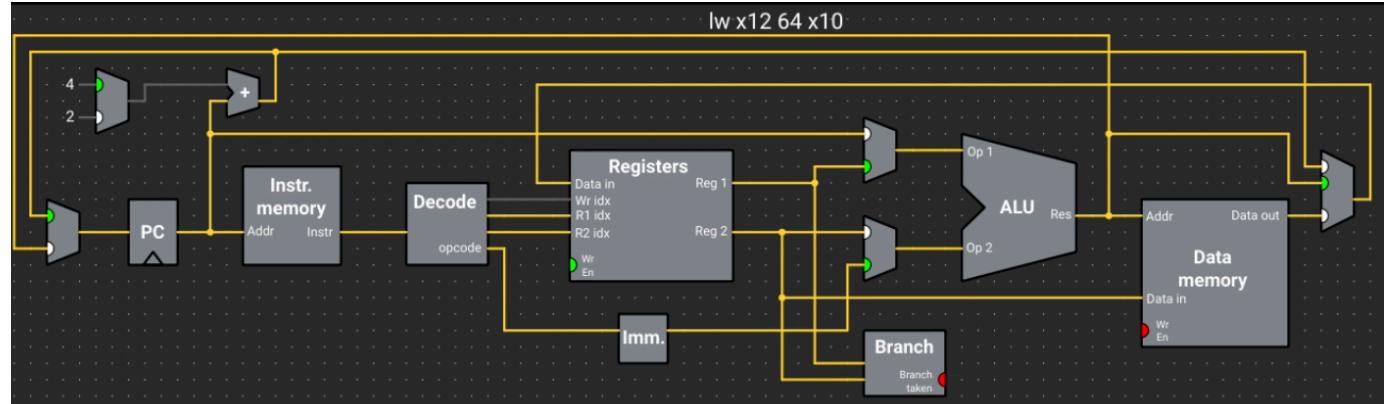


The 'little-endian' and 'big-endian' terminology which is used to denote the two approaches [to addressing memory] is derived from Swift's *Gulliver's Travels*. The inhabitants of Lilliput, who are well known for being rather small, are, in addition, constrained by law to break their eggs only at the little end. When this law is imposed, those of their fellow citizens who prefer to break their eggs at the big end take exception to the new rule and civil war breaks out. The big-endians eventually take refuge on a nearby island, which is the kingdom of Blefuscus. The civil war results in many casualties.

Read: Holy Wars and a Plea For Peace, D. Cohen

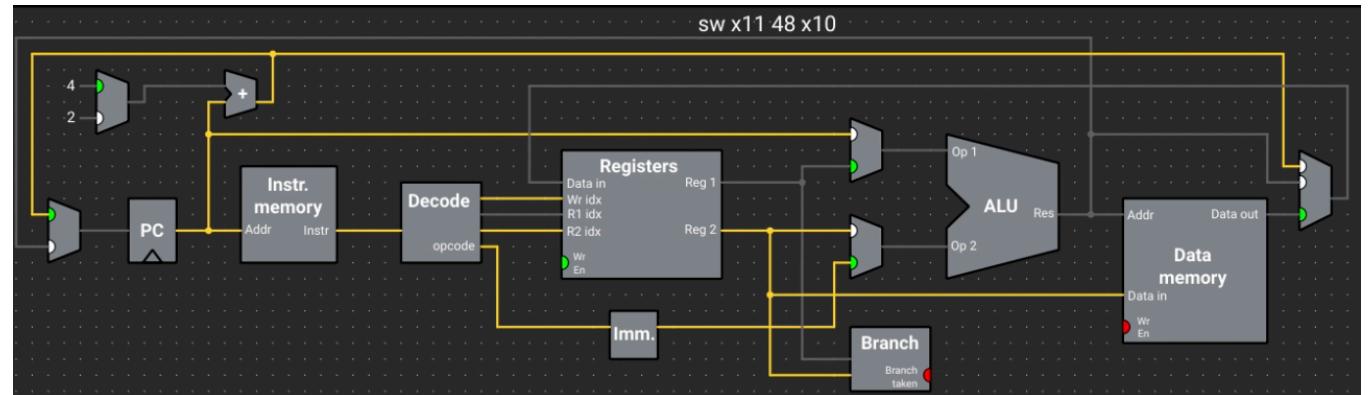
Load and store instructions

lw a2,0x40(a0)



Offset expressed as immediate,
add to base to compute memory address
read/write contents at that address

sw a1,0x30(a0)



Let there be light

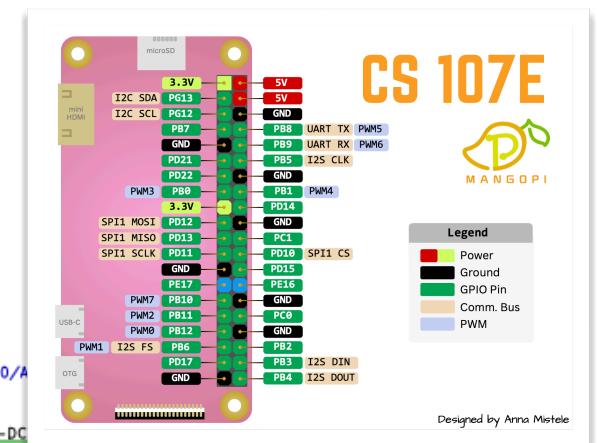
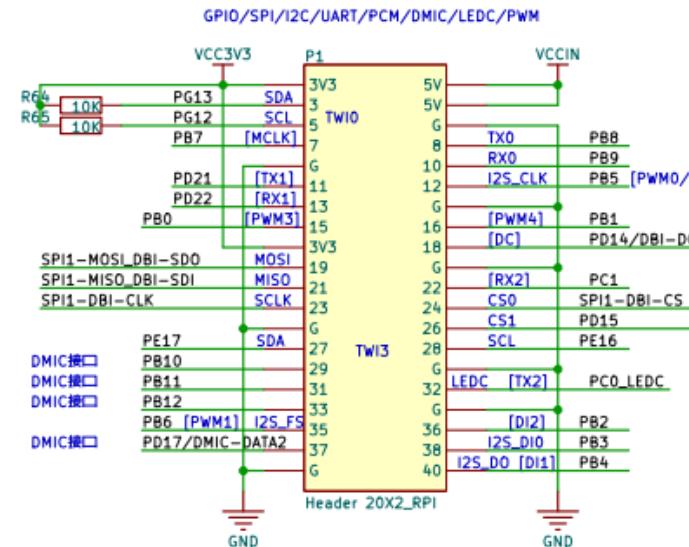
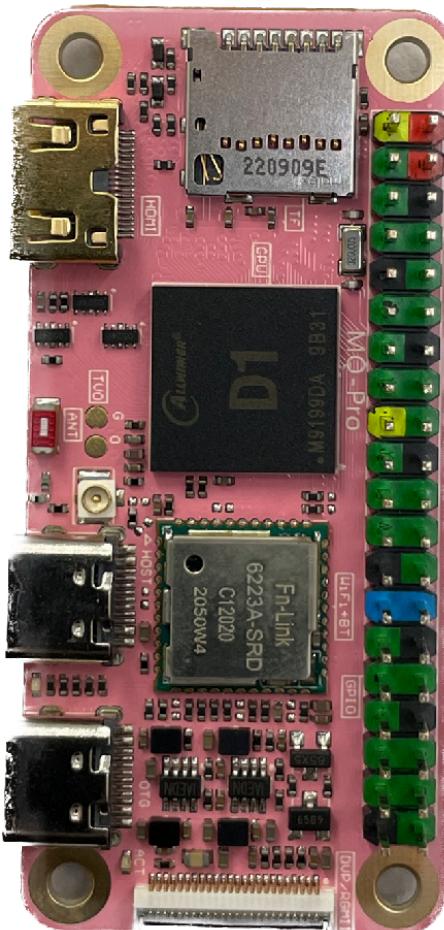


Computer *peripheral* interfaces to outside world

GPIO pins are peripherals

Let's learn how to control a GPIO pin with code!

Mango Pi GPIO



9.7 GPIO

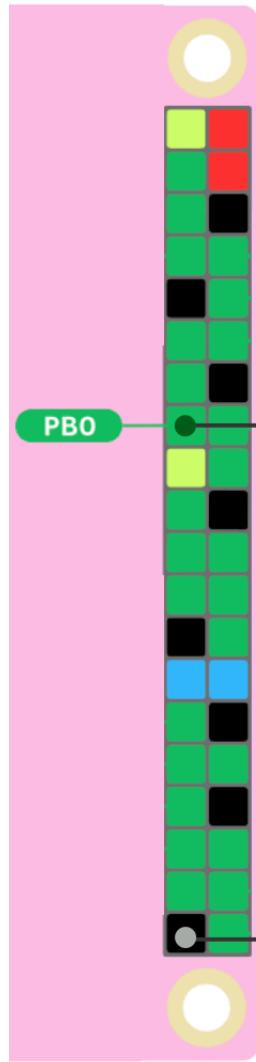
9.7.1 Overview

The general purpose input/output (GPIO) is one of the blocks controlling the chip multiplexing pins. The D1-H supports 6 groups of GPIO pins. Each pin can be configured as input or output and these pins are used to generate input signals or output signals for special purposes.

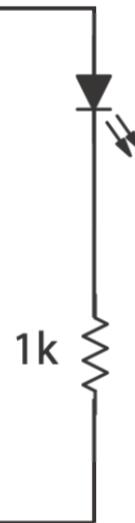
The Port Controller has the following features:

- 6 groups of ports (PB, PC, PD, PE, PF, PG)
- Software control for each signal pin
- Data input (capture)/output (drive)
- Each GPIO peripheral can produce an interrupt

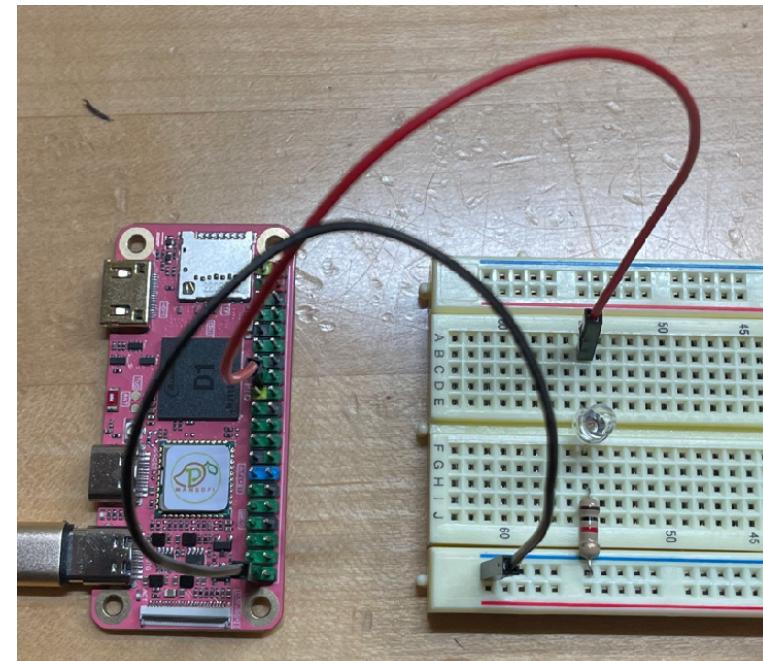
Connect LED to GPIO PB0



Set/clear GPIO will change output voltage
set 1 -> 3.3V
clear 0 -> 0.0V



Complete circuit
(to ground pin)



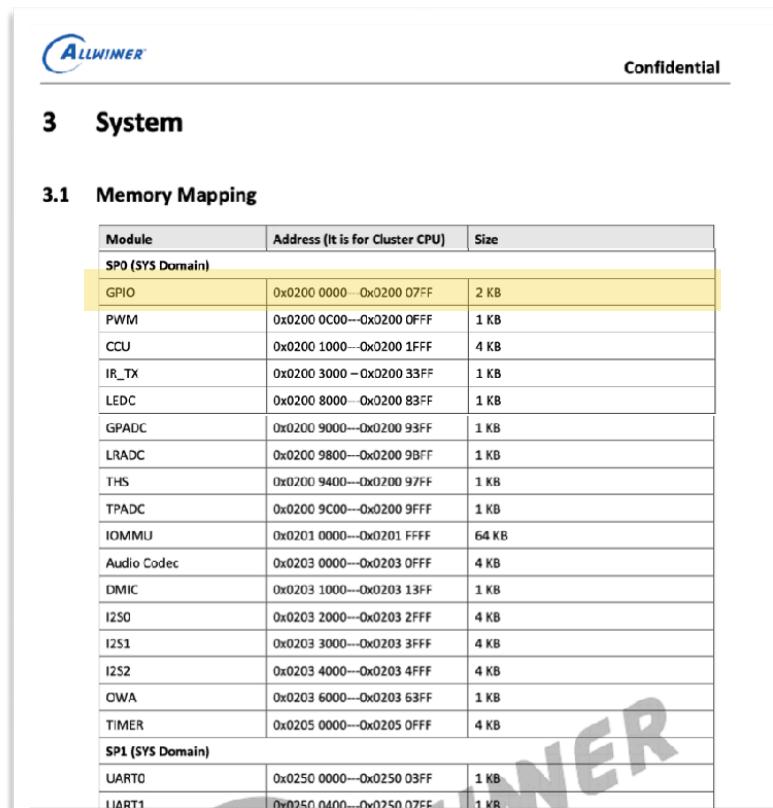
Q: What is purpose of resistor in this circuit?

Memory Map

Peripheral registers are mapped into address space

Read/write to memory address controls peripheral

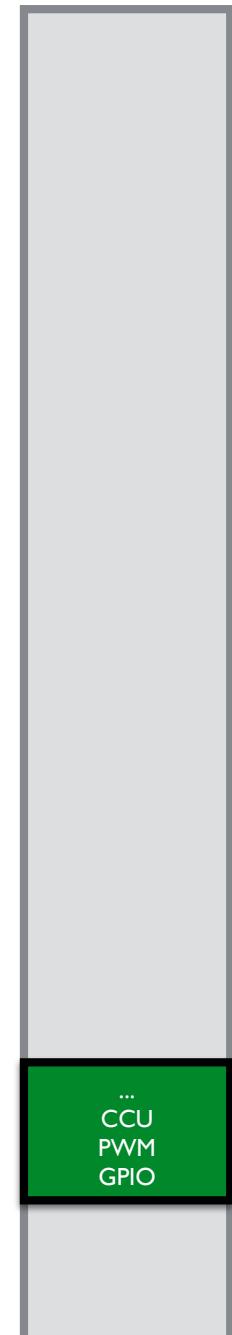
This is
"memory-mapped IO"
(MMIO)



The screenshot shows a table titled "3 System" under "3.1 Memory Mapping". The table lists memory modules, their addresses, and sizes. The "SP0 (SYS Domain)" section includes GPIO, PWM, CCU, IR_TX, LEDC, GPADC, LRADC, THS, TPADC, IOMMU, Audio Codec, DMIC, I2S0, I2S1, I2S2, CWA, and TIMER. The "SP1 (SYS Domain)" section includes UART0 and UUART1. The "Address (It is for Cluster CPU)" column shows ranges like 0x0200 0000 - 0x0200 07FF for GPIO and 0x0250 0000 - 0x0250 03FF for UUART1. The "Size" column shows sizes like 2 KB for GPIO and 1 KB for UUART1.

Module	Address (It is for Cluster CPU)	Size
SP0 (SYS Domain)		
GPIO	0x0200 0000 – 0x0200 07FF	2 KB
PWM	0x0200 0C00 – 0x0200 OFFF	1 KB
CCU	0x0200 1000 – 0x0200 1FFF	4 KB
IR_TX	0x0200 3000 – 0x0200 33FF	1 KB
LEDC	0x0200 8000 – 0x0200 83FF	1 KB
GPADC	0x0200 9000 – 0x0200 93FF	1 KB
LRADC	0x0200 9800 – 0x0200 9BFF	1 KB
THS	0x0200 9400 – 0x0200 97FF	1 KB
TPADC	0x0200 9C00 – 0x0200 9FFF	1 KB
IOMMU	0x0201 0000 – 0x0201 FFFF	64 KB
Audio Codec	0x0203 0000 – 0x0203 0FFF	4 KB
DMIC	0x0203 1000 – 0x0203 13FF	1 KB
I2S0	0x0203 2000 – 0x0203 2FFF	4 KB
I2S1	0x0203 3000 – 0x0203 3FFF	4 KB
I2S2	0x0203 4000 – 0x0203 4FFF	4 KB
CWA	0x0203 6000 – 0x0203 63FF	1 KB
TIMER	0x0205 0000 – 0x0205 OFFF	4 KB
SP1 (SYS Domain)		
UART0	0x0250 0000 – 0x0250 03FF	1 KB
UUART1	0x0250 0400 – 0x0250 07FF	1 KB

Ref: **DI-H User Manual p.45**



0x02000000

9.7.4 Register List

Module Name	Base Address
GPIO	0x02000000

Register Name	Offset	Description
PB_CFG0	0x0030	PB Configure Register 0
PB_CFG1	0x0034	PB Configure Register 1
PB_DAT	0x0040	PB Data Register
PB_DRV0	0x0044	PB Multi_Driving Register 0
PB_DRV1	0x0048	PB Multi_Driving Register 1
PB_PULL0	0x0054	PB Pull Register 0
PC_CFG0	0x0060	PC Configure Register 0
PC_DAT	0x0070	PC Data Register
PC_DRV0	0x0074	PC Multi_Driving Register 0
PC_PULL0	0x0084	PC Pull Register 0

Access Configure Register to select pin function

Access Data Register to change pin value

9.7.3.2 GPIO Multiplex Function

Table 9-21 to Table 9-26 show the multiplex function pins of the D1-H.



NOTE

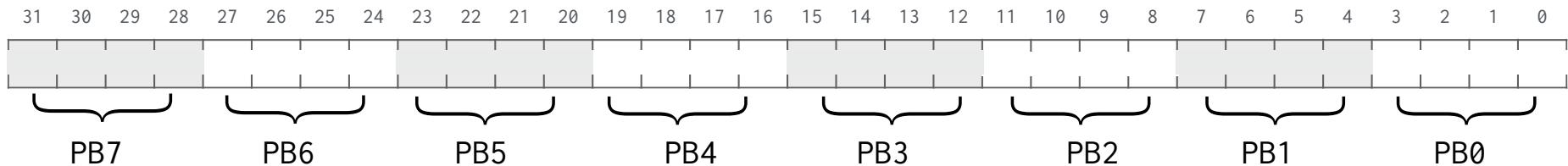
For each GPIO, Function0 is input function; Function1 is output function; Function9 to Function13 are reserved.

Table 9-21 PB Multiplex Function

GPIO Port	Function 2	Function 3	Function 4	Function 5	Function 6	Function 7	Function 8	Function 14
PB0	PWM3	IR-TX	TWI2-SCK	SPI1-WP/DBI-TE	UART0-TX	UART2-TX	OWA-OUT	PB-EINT0
PB1	PWM4	I2S2-DOUT3	TWI2-SDA	I2S2-DIN3	UART0-RX	UART2-RX	IR-RX	PB-EINT1
PB2	LCD0-D0	I2S2-DOUT2	TWI0-SDA	I2S2-DIN2	LCD0-D18	UART4-TX		PB-EINT2
PB3	LCD0-D1	I2S2-DOUT1	TWI0-SCK	I2S2-DIN0	LCD0-D19	UART4-RX		PB-EINT3
PB4	LCD0-D8	I2S2-DOUT0	TWI1-SCK	I2S2-DIN1	LCD0-D20	UART5-TX		PB-EINT4

GPIO Configure Register

PB Config0 @0x02000030



4 bits per GPIO pin

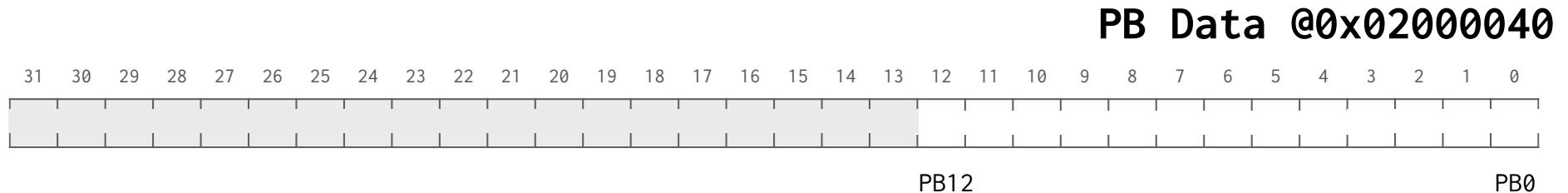
32-bit register stores
config for 8 pins

Pin select = 4 bits (16 options)
Input (0), Output (1),
Alt2-Alt8, 9-13 reserved,
Interrupt (14), Disabled (15)

Offset: 0x0030			Register Name: PB_CFG0																
Bit	Read/Write	Default/Hex	Description																
3:0	R/W	0xF	<table><tr><td>PB0_SELECT</td><td></td></tr><tr><td>PB0 Select</td><td></td></tr><tr><td>0000:Input</td><td>0001:Output</td></tr><tr><td>0010:PWM3</td><td>0011:IR-TX</td></tr><tr><td>0100:TWI2-SCK</td><td>0101:SPI1-WP/DBI-TE</td></tr><tr><td>0110:UART0-TX</td><td>0111:UART2-TX</td></tr><tr><td>1000:OWA-OUT</td><td>1001:Reserved</td></tr><tr><td>1110:PB-EINT0</td><td>1111:IO Disable</td></tr></table>	PB0_SELECT		PB0 Select		0000:Input	0001:Output	0010:PWM3	0011:IR-TX	0100:TWI2-SCK	0101:SPI1-WP/DBI-TE	0110:UART0-TX	0111:UART2-TX	1000:OWA-OUT	1001:Reserved	1110:PB-EINT0	1111:IO Disable
PB0_SELECT																			
PB0 Select																			
0000:Input	0001:Output																		
0010:PWM3	0011:IR-TX																		
0100:TWI2-SCK	0101:SPI1-WP/DBI-TE																		
0110:UART0-TX	0111:UART2-TX																		
1000:OWA-OUT	1001:Reserved																		
1110:PB-EINT0	1111:IO Disable																		

Ref: [DI-H User Manual p.1097](#)

GPIO Data Register



I bit per GPIO pin

Value is 1 if high, 0 low

9.7.5.3 0x0040 PB Data Register (Default Value: 0x0000_0000)

Offset: 0x0040			Register Name: PB_DAT
Bit	Read/Write	Default/Hex	Description
31:13	/	/	/
12:0	R/W	0x0	PB_DAT If the port is configured as the input function, the corresponding bit is the pin state. If the port is configured as the output function, the pin state is the same as the corresponding bit. The read bit value is the value set up by software. If the port is configured as a functional pin, the undefined value will be read.

Ref: DI-H User Manual p.1098

Using xfel

BOOTROM of Mango Pi starts in "FEL" by default
(Firmware Exchange Loader)

FEL will listen on USB port

Connect laptop to Mango Pi USB OTG port

Run `xfel` on your laptop will communicate with FEL on Pi

Can use `xfel` to peek and poke memory addresses, load and execute programs, and more

```
$ xfel write32 0x02000030 0x1
$ xfel write32 0x02000040 0x1
```

on.s

```
lui    a0,0x2000  
addi   a1,zero,1  
sw     a1,0x30(a0)  
  
sw     a1,0x40(a0)  
  
loop:  
      j  loop
```

} Select output fn pin PB0

} Set pin PB0 data value to 1

} Loop infinitely

Reminders:

PB CFG0 register @ 0x02000030

PB DATA register @ 0x02000040

Build and execute

```
$ riscv64-unknown-elf-as on -o on  
  
$ riscv64-unknown-elf-objcopy on.o on.bin -O binary  
  
$ mango-run on.bin  
    xfel ddr d1  
    xfel write 0x40000000 on.bin  
    xfel exec 0x40000000
```

- 1) assembler translates assembly insns to machine encoding
- 2) objcopy extracts raw binary
- 3) mango-run: xfel init dram, load program, execute it

blink.s

```
lui      a0,0x2000
addi    a1,zero,1
sw      a1,0x30(a0)      # config PB0 as output

loop:
xori    a1,a1,1          # xor ^ 1 invert bit 0
sw      a1,0x40(a0)      # flip bit on<->off

        lui    a2,0x3f00      # busy loop wait
delay:
addi    a2,a2,-1
bne    a2,zero,delay

j      loop               # repeat forever
```

Key concepts so far

Translation between assembly and machine encoding
(assembler/disassembler)

Mango Pi uses little-endian memory organization

General purpose IO (GPIO), peripheral registers, MMIO

Tools to build, load, execute a program

Resources to keep handy

RISC-V one-page guide

Ripes simulator

Mango Pi pinout

DI-H User Manual