TPU Instruction Set Architecture v1.5

Revision 6

May 23rd 2016

Domipheus Labs

http://labs.domipheus.com

Contents

Version History

Version	Description	Date
1.0	Initial	
1.1	Added Save PC, Save Status, add immediate unsigned, sub immediate unsigned, Jump conditional to relative offset	July 31 2015
1.2	References to jumps replaced with branches. Branch conditionals with new instruction forms. Read and Write now have a signed offset in the encoding.	August 3 2015
1.3	Byte addressing modes for read/write & endianness	September 21st 2015
1.4	Exception handling model and instructions	October 1st 2015
1.5	INT instruction, fixing of flag0 bit of add and sub instructions from reserved to fixed 0. Branch immediate was removed, in it's place we now have branch to immediate relative offset.	May 1st 2016
1.5 r6	Subi.u and addi.u: removed .u, clarified immediate is always unsigned	May 23rd 2016

Known Issues

Arithmetic does not set correct overflow status flags

The Addition and Subtraction instructions, of signed variants, do not correctly set overflow flags.

Latencies

Latencies are not fully known until CPU design is finalized; however, all instructions apart from memory read and write take a fixed amount of cycles. Memory read and write add additional latencies.

Interrupt model in flux

Currently, interrupts can automatically disable on hit, but this is not fully exposed/documented yet.

TPU is not pipelined.

Each instruction takes at minimum 7 clock cycles.

Endianness

TPU is currently big endian. 16 bit values are stored from bit 15 to bit 0 as MSB to LSB high byte to low byte.

When reading/writing 16-bit words to and from memory, the most significant byte will be written to the smaller address, the least significant byte written to address+1.

Glossary

Op	Op Code
Rd	Destination Register
F	Function flags
Ra	Source Register A
Rb	Source Register B
Imm	Immediate Value
Fs	Signed Flag
R	Reserved
X	Undefined
C	Conditiona flags

Instruction Forms

Definitions

- Form RRR
  - Destination and two source registers
- Form RRs
  - Two source registers, optional immediate
- Form RRd
  - One destination and one source register, optional immediate
- Form CRsI
  - Ass RRd, however, rD is used as a constant flag section.
- Form CRR
  - Condition flags and two source registers
- Form R
  - A single source register
- Form RRImm
  - Destination, Source and 4-bit immediate.
- Form RImm
  - Destination register with 8-bit immediate value
- Form Imm
  - 8-bit immediate value, can optionally use [11:9] to extend the immediate.
- Form RRImm
  - Destination reg, source reg and 4-bit immediate

All forms have various flag and unused bit spaces that instruction operations may use.

Layout

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RRR	opcode				rD			F	rA			rB			Unused	
RRs	opcode				Imm[4:2]			F	rA			rB			Imm[1:0]	
RRd	opcode				rD			F	rA			5-bit immediate				
CRsI	opcode				C			F	rA			5-bit immediate				
CRR	opcode				C			F	rA			rB			Unused	
RRImm	opcode				rD			F	rA			4-bit Immediate			?	
R	opcode				rD			F	Unused							
RImm	opcode				rD			F	8-bit Immediate Value							
Imm	opcode				Optional Imm			F	8-bit Immediate Value							

Instruction Listing

add.s, add.u

Add signed or unsigned

Instruction Form RRR

15	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
0	**0**	**0**	**0**	Rd	Rd	Rd	**S**	Ra	Ra	Ra	Rb	Rb	Rb	R	**0**

Regs[Rd] = Regs[Ra] + Regs[Rb]

The bit S indicates whether the addition operation is signed. If S is 1, the operands are taken as signed integer values, unsigned otherwise.

addi

Add immediate

Instruction Form RRImm

15	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
0	**0**	**0**	**0**	Rd	Rd	Rd	**0**	Ra	Ra	Ra	Imm	Imm	Imm	Imm	**1**

Regs[Rd] = Regs[Ra] + Imm

Imm is always considered unsigned

sub.s, sub.u

Subtract signed or unsigned

Instruction Form RRR

15	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
0	**0**	**0**	**1**	Rd	Rd	Rd	**0**	Ra	Ra	Ra	Rb	Rb	Rb	R	**0**

Regs[Rd] = Regs[Ra] - Regs[Rb]

The bit S indicates whether the addition operation is signed. If S is 1, the operands are taken as signed integer values, unsigned otherwise.

subi

Add immediate

Instruction Form RRImm

15	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*

0	**0**	**0**	**1**	Rd	Rd	Rd	**0**	Ra	Ra	Ra	Imm	Imm	Imm	Imm	**1**

Regs[Rd] = Regs[Ra] - Imm

Imm is always considered unsigned

or

Bitwise OR

Instruction Form RRR

15	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
0	**0**	**1**	**0**	Rd	Rd	Rd	**0**	Ra	Ra	Ra	Rb	Rb	Rb	R	R

Regs[Rd] = Regs[Ra] or Regs[Rb]

xor

Bitwise XOR

Instruction Form RRR

15	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
0	**0**	**1**	**1**	Rd	Rd	Rd	**0**	Ra	Ra	Ra	Rb	Rb	Rb	R	R

Regs[Rd] = Regs[Ra] xor Regs[Rb]

and

Bitwise AND

Instruction Form RRR

15	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
0	**1**	**0**	**0**	Rd	Rd	Rd	**0**	Ra	Ra	Ra	Rb	Rb	Rb	R	R

Regs[Rd] = Regs[Ra] and Regs[Rb]

not

Bitwise NOT

Instruction Form RRd

15	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
0	**1**	**0**	**1**	Rd	Rd	Rd	**0**	Ra	Ra	Ra	**0**	**0**	**0**	R	R

Regs[Rd] = not Regs[Ra]

read, read.w, read.b

Memory Read

Instruction Form RRd

15	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
0	**1**	**1**	**0**	Rd	Rd	Rd	**B**	Ra	Ra	Ra	Imm				

Regs[Rd] = Memory[Regs[Ra] + Imm]

Reads a value from memory at the specified location into the destination register. When B=1, a byte is read. If B=0, a 16-bit value is read. Byte addressing applies throughout.

The Immediate offset is considered a signed value.

write, write.w, write.b

Memory Write

Instruction Form RRsW

15	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
0	**1**	**1**	**1**	Imm	Imm	Im m	**B**	Ra	Ra	Ra	Rb	Rb	Rb	Imm	Im m

Memory[Regs[Ra] + Imm] = Regs[Rb]

Writes the value in Rb into memory at the specified location. When B=1, a byte is written (lower half of register Rb). If B=0, a 16-bit value is written. Byte addressing applies throughout.

The Immediate offset is considered a signed value.

load.h, load.l

Load Immediate High, Load Immediate Low

Instruction Form RImm

15	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
1	**0**	**0**	**0**	Rd	Rd	Rd	**LF**	Im	Im	Im	Im	Im	Im	Im	Im

If LF = 1 then

Regs[Rd] = 0x00FF & Im

else

Regs[Rd] = 0xFF00 & Im<<8

End if

cmp.s, cmp.u

Compare Integers

Instruction Form RRR

15	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
1	**0**	**0**	**1**	*Rd*			*Fs*	*Ra*			*Rb*			R	R

Regs[***Rd***] = *compare* (Regs[***Ra***] , Regs[***Rb***], Fs)

Compares integer values within registers ***Ra*** and ***Rb***, placing a result bit field in ***Rd***.

If ***Fs*** is set, comparisons are treated as signed integers. The result bit field written to ***Rd*** is defined as follows. The result of the comparison goes into the associated bit of the register, a set bit indicating true.

15	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
X	**Ra = Rb**	**Ra > Rb**	**Ra < Rb**	**Ra = 0**	**Rb = 0**	X	X	X	X	X	X	X	X	X	X

shl

Shift Left Logical from Register

Instruction Form RRR

15	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
1	**0**	**1**	**0**	Rd	Rd	Rd	**0**	Ra	Ra	Ra	Rb	Rb	Rb	R	R

Regs[Rd] = Regs[Ra] << Regs[Rb]

shr

Shift Right Logical from Register

Instruction Form RRR

15	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
1	**0**	**1**	**1**	Rd	Rd	Rd	**0**	Ra	Ra	Ra	Rb	Rb	Rb	R	R

Regs[Rd] = Regs[Ra] >> Regs[Rb]

br

branch to register location

Instruction Form RRR

15	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
1	**1**	**0**	**0**	0	0	0	**0**	Ra	Ra	Ra	0	0	0	R	R

PC = Regs[Ra]

biro

Branch to immediate relative offset

Instruction Form Imm

15	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
1	**1**	**0**	**0**	Im	Im	Im	**1**	Im	Im	Im	Im	Im	Im	Im	Im

Immediate value is treated as signed.

PC = PC + (Im << 1);

br.eq, br.az, br.bz, br.anz, br.bnz, br.gt, br.lt

Branch Conditional

Instruction Form CRR

15	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
1	**1**	**0**	**1**	C_2	C_1	C_0	**0**	Ra			Rb			0	0

If **C** matches with condition bits written by a CMP instruction stored in Reg[**Ra**] then

PC = Reg[**Rb**]

Table of **C** bits to condition mappings.

C_2, C_1, C_0	Condition
0, 0, 0	EQ
0, 0, 1	Ra = 0
0, 1, 0	Rb = 0
0, 1, 1	Ra != 0
1, 0, 0	Rb != 0
1, 0, 1	Ra > Rb
1, 1, 0	Ra < Rb

bro.eq, bro.az, bro.bz, bro.anz, bro.bnz, bro.gt, bro.lt

Branch conditional to relative offset

Instruction Form CRsI

15	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
1	**1**	**0**	**1**	C_2	C_1	C_0	**1**	Ra			Imm_4	Imm_3	Imm_2	Imm_1	Imm_0

Imm is signed.

If **C** matches with condition bits written by a CMP instruction stored in Reg[**Ra**] then

PC = PC + Imm

Table of **C** bits to condition mappings.

C_2, C_1, C_0	Condition
0, 0, 0	EQ
0, 0, 1	Ra = 0
0, 1, 0	Rb = 0
0, 1, 1	Ra != 0
1, 0, 0	Rb != 0
1, 0, 1	Ra > Rb
1, 1, 0	Ra < Rb

spc

Save PC

Instruction Form RRR

15	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
1	**1**	**1**	**0**	Rd	Rd	Rd	**0**	0	0	0	0	0	0	**0**	**0**

Regs[Rd] = PC

sstatus

Save Status

Instruction Form RRR

15	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
1	**1**	**1**	**0**	Rd	Rd	Rd	**0**	0	0	0	0	0	0	**0**	**1**

Regs[Rd] = Status

gief

Get Interrupt Event Field

Instruction Form RRR

15	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
1	**1**	**1**	**0**	Rd	Rd	Rd	**1**	0	0	0	0	0	0	**0**	**0**

Regs[Rd] = Interrupt Data Register Contents

bbi

Branch back from Interrupt

Instruction Form RRR

15	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
1	**1**	**1**	**0**	0	0	0	**1**	0	0	0	0	0	0	**0**	**1**

Branches to the PC which was to be executed next before the last interrupt was encountered.

ei

Enable Interrupt

Instruction Form RRR

15	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
1	**1**	**1**	**0**	Rd	Rd	Rd	**1**	0	0	0	0	0	0	**1**	**0**

di

Disable Interrupt

Instruction Form RRR

15	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
1	**1**	**1**	**0**	Rd	Rd	Rd	**1**	0	0	0	0	0	0	**1**	**1**

int

Fire Interrupt programatically

Instruction Form Imm

15	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
1	**1**	**1**	**0**	**0**	**0**	**0**	**0**	Imm	Imm	Imm	Imm	Imm	Imm	**1**	**0**

Saves the current PC+2 value for use in the bbi instruction, loads the Interrupt Event Field with the immediate value in [7:2], and branches to the interrupt vector.

Due to this, it is advised that Interrupt handlers treat Interrupt Event Field values of 0-63U as software interrupts, that can be invoked from user code.

The interrupt handler will be called, regardless of interrupts being enabled or not. Upon an int instruction execute, interrupts will be automatically disabled.