G6A-RISC

General Description

G6A-RISC is an experimental relay based computer for learning and educational purposes. It starts from the knowledge available from previously built 'modern' relay based computers, but aims at an easier to use instruction set, with fixed length instructions, constant instruction execution time, and a cleaner hardware architecture.

It is based on the Harvard architecture with separated program and data memory, with 16 bit wide registers and addressable memory space. Despite being labeled 'RISC', it is not a load/store architecture, as ALU operations on memory are allowed.

Binary Instruction Formats

Instruction encodings are fixed 16 bit wide and they are defined by a leading 2 bits 'Mode' field followed by a 3 bits 'Opcode' and a 11 bit operands encoding depending on Mode.

General instruction patterns:

Type	Mode		Opcode			Operand Encoding											Description
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
M	00		op			Ri			An		s	imm (5 bits)					Indexed Memory Addressing
ZP	01		op			Ri			fn		s	imm (5 bits)					Direct Memory Addressing
R	10		op			Ri/CC			fn		Rj			Rk			Three registers
I	11		op			Ri			fn		0	imm (5 bits)					Immediate, Branch

op: 3 bit opcode
fn: 2 bit function code
s : 1 bit field indicating that the instruction is a memory store
Ri: Destination register
Rj, Rk: source registers
CC: Condition code
An: Address register (Register An is encoded as n+1. n = 0 is reserved for extended instructions)

Extended instructions:

Instr	Mode		Opcode			Operand Encoding											Description
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
pfx	00		000			immediate (11 bits)											Prefix
hlt	00		001			xxx			00		0	x	x	xxx			Halt
lp	00		011			Ri			00		0	x	x	Rk			Load from Program Memory
r0a	00		110			Ri			00		0	imm (5 bits)					R0 accumulate
r1a	00		110			Ri			00		0	imm (5 bits)					R1 accumulate

The 'nop' instruction is emulated with an all zeros encoding

Registers

Register	Alt Name	Description
R0	-	16 bit, General Purpose
R1	-	16 bit, General Purpose
R2	-	16 bit, General Purpose
R3	-	16 bit, General Purpose
R4	A0	16 bit, General Purpose, Address Register
R5	A1	16 bit, General Purpose, Address Register
R6	A2, LR	16 bit, General Purpose, Address Register, Link Register
R7	PC	16 bit, Program Counter
PFR	-	11 bit, Prefix Register

All registers are 16 bit. ALU operations are 16 bit. 8 bit operations are not supported.
Registers R1 through R6 are general purpose.
Registers R4 through R7 are used in M-type instructions as base address.
Register R6 is used as the link register for the 'brl' instruction.

PC is the Program Counter. It can be accessed as Register 7 with regular instructions. Writing to it causes program execution to jump to the specified address. Memory reads with PC as the base address are illegal. Use the 'lp' instruction instead to read data from program memory.

PFR is the prefix register. It is written by the prefix instruction and implicitly used by type I instructions.

There's no Stack Pointer register. Subroutine returns are handled with the link register. Stack frames can be explicitly created with regular instructions.

Status Register

Register	Description
Status T C Z	Status Register T: Condition flag, result of a compare instruction C, Z: Carry, Zero flags, result of ALU operations
RR D[3..0]	Rotate Digit Contains the four bits being rotated by 'rr4', 'rl4' instructions

Compare instructions compare two operands for a specified condition code and set T to 1 if the condition was met or 0 otherwise. C and Z flags are updated acordingly.

Most ALU arithmetic and logical instructions set C and Z according to the result. Additionally, the Z flag or C flag is copied to T. For example, the 'add' instruction will set C to 1 if there was a carry, and both Z and T to 1 if the result was zero.

Conditional instructions such as 'set', 'sef' and 'sel' and 'bt+' use the T flag as the condition to watch.

Addressing Modes

There are four possible addressing modes, with the following meanings:

Type	Denomination	Machine Operator	Assembly	Description
M	Indexed Memory	mem(An|K)	[An, K]	Contents of memory at the address determined by performing a bitwise or of An with K
ZP	Direct Memory load (also called Zero Page)	mem(K)	[K]	Contents of memory at address pointed by K
R	Register	Ri	Ri	Register content
I	Immediate	K	K	Constant value

Addressing modes relate with instruction Types

Type	Addressing mode	Source 1	Source 2	Destination	Assembly Example
M	Indexed Memory load	Ri	mem(An|K)	Ri	add [An, K], r0
	Indexed Memory store (1)	Ri	mem(An|K)	mem(An|K)	add Ri, [An, K]
ZP	Direct Memory load	Ri	mem(K)	Ri	add [K], ri
	Direct Memory store (1)	Ri	mem(K)	mem(K)	add ri, [K]
R	Register (2) (3) (4)	Rj	Rk	Ri/CC	add Rj, Rk, Ri cmp.CC Rj, Rk mov Rk, Ri sl1 Rj, Ri
I	Immediate	Ri	K	Ri	add K, Ri
	Branch			PC	b+ .label (same as 'add K, PC')

(1) Store instructions are expressed in assembler by placing the memory operand in the last position. Loads have the register destination as the last operand. Execution order will always be 'Source1' operator 'Source2' -> 'Destination' regardless of loads or stores. This is particularly relevant for non-commutative operations such as subtractions.

(2) The 'set', 'sef', instructions ignore Source1. The 'mov' instruction ignores Source1 for loads, and ignores Source2 for stores.

(3) The 'sl1, 'sl4', 'sr1', 'sr4', instructions ignore Source2.

(4) The 'cmp' and 'cpc' instructions do not set a destination register. for R-Type instructions, the destination register field is used to encode the condition code to test, which acts as a third source operand.

All instructions can be conceptually considered to contain the following fields:

Operation	Source1	Source2	Destination

However, assembly language Instructions may not explicitly express all of them. For example, M-Type, ZP-Type and I-Type instructions do not need Source1 because it is implicit.

On the following sections, we will use the terms 'Source1', 'Source2' and 'Destination' as synonymous of the operators found in assembly instructions, independently of Addressing Mode. Therefore, is it useful to refer to the table above to determine the relevant operators for the Addressing Mode in use.

Instruction Opcodes and Encodings

		M-Type			ZP-Type								R-Type				I-Type				(J-Type)			
type		00			01								10				11				11			
s		0		1	0				1				-				-				-			
fn		00	An		00	01	10	11	00	01	10	11	00	01	10	11	00	01	10	11	00	01	10	11
op	000	pfx			sr1	rr1	sr4	rr4	sr1	rr1	sr4	rr4	sr1	rr1	sr4	rr4	sr1	rr1	sr4	rr4	-	-	-	-
	001	hlt			sl1	rl1	sl4	rl4	sl1	rl1	sl4	rl4	sl1	rl1	sl4	rl4	sl1	rl1	sl4	rl4	-	-	-	-
	010	-	cmp	(1)	cmp	cpc	set	sef	(1)	(1)	set	sef	cmp	cpc	set	sef	cmp	cpc	set	sef	-	-	-	-
	011	lp	mov	mov	mov	mvl	-	sel	mov	mvl	-	sel	mov	mvl	-	sel	mov	mvl	-	sel	j	jl	-	-
	100	-	add	add	add	sub	adt	sbt	add	sub	adt	sbt	add	sub	adt	sbt	add	sub	adt	sbt	b+	b-	bt+	bt-
	101	-	adc	adc	adc	sbc	adf	sbf	adc	sbc	adf	sbf	adc	sbc	adf	sbf	adc	sbc	adf	sbf	-	-	bf+	bf-
	110	r0a	dad	dad	dad	dsb	or	xor	dad	dsb	or	xor	dad	dsb	or	xor	dad	dsb	or	xor	-	-	-	-
	111	r1a	dac	dac	dac	dsc	and	rsb	dac	dsc	and	rsb	dac	dsc	and	rsb	dac	dsc	and	rsb	-	-	-	-

op: 3 bit opcode for the instruction type
fn: 2 bit function code, for M-Type instructions this is an address register
s : 1 bit field indicating that the instruction is a memory store

(1) Instruction slot not available, will cause undocumented behaviour.

(*) J-Type instructions are regular I-Type instructions that change their name when used with the PC as the destination register.

(*) M-Type instructions with op=000 or fn=00 have special meanings.

Assembly Instruction Format

Assembly instructions are described with a 3 character mnemonic following by 2 or 3 operands separated by commas. By convention the last operand is always the destination for instructions producing a result.

* The P-Type prefix instruction takes a single 11 bit immediate operand.

* I-Type instructions take 2 operands, an immediate 5 bit value and a destination register. I-type instructions may have a different meaning, or produce undocumented behaviour, when used with the PC register.

* R-Type instructions take 3 register operands, operand 1 and 2 are the Source operands, operand 3 is the Destination.

* M-Type and ZP-Type instructions take 2 operands, a register operand as and an indexed memory operand. Bit 's' determines whether the operation is a load or a store, this is specified in assembler by the order of operands. The last operand is the destination. For the M-Type the effective address is computed by adding the given address register to the 5-bit immediate. For the ZP-Type instructions, the immediate value is used.

* As described later, all instructions with an immediate field can be prefixed in order to extend the constant range up to 16 bits.

<u>Program example1</u>

Assume a stack based machine where the data stack is pointed by register 'a0'. The stack grows down the memory addresses.

Multiply using the 'booth' algorithm. End when the multiplicand is zero. The core multiplication uses up to 113 cycles but will be much faster for small multiplicands

```
        mov 100, a0         // assume 100 is the top of the stack address
        mov [a0, 0], r1     // get multiplier
        mov [a0, 1], r2     // get multiplicand
        mov 0, r0           // set result to zero
.LMulHi
        cmp.eq 0, r2        // compare multiplicand with zero
        bt+ .LMulDone       // branch if zero
        sr1 r2, r2          // shift right the multiplicand
        set r1, r3          // set r3 to the multiplier or zero
        add r0, r3, r0      // add multiplier (or zero)
        sl1 r1, r1          // shift multiplier left
        b- .LMulHi          // next iteration
.LMulDone
        add 1, a0           // increment the data stack pointer
        mov r0, [a0, 0]     // store the result on top of the stack
```

<u>Program example 2</u>

Similar to the previous example but with a constant execution time

Multiply using the 'booth' algorithm. Constant execution time. The core multiplication uses 96 cycles

```
        mov 100, a0         // assume 100 is the top of the stack address
        mov [a0, 0], r1     // get multiplier
        mov [a0, 1], r2     // get multiplicand
        add 1, a0           // increment stack address
        mov 0, r0
        mov r0, [a0, 0]     // set initial result to zero
        mov 16, r0          // initialise counter
.LMulHi
        sr1 r2, r2          // shift right the multiplicand
        set r1, r3          // conditionally set r3 to the multiplier
        add r3, [a0, 0]     // accumulate the result
        sl1 r1, r1          // shift multiplier left
        sub 1, r0           // decrement counter
        bt- .LMulHi         // next iteration
.LMulDone
        // done, the stack pointer is already incremented
        // and the result in the right memory location
```

Prefix, Prefixed Instructions

Prefixed instructions are assembler emulated instructions of type M, ZP, or I that are made of core instructions preceded by a 'pfx' instruction. The 'pfx' instruction contains an 11 bit immediate field, that extends the 5 bit immediate field of the following instruction by supplying the upper 11 bits. This provides a full 16 bit immediate range to the prefixed instruction.

The following *non-exhaustive* list shows several examples of prefix instruction transformations:

Core Instruction	Prefixed Instruction	Description
Immediate		
add 257, Ri	pfx 8 add 1, Ri	Add with long immediate. The immediate value does not fit in 5 bits, thus a pfx instruction is inserted.
and 255, Ri	pfx 8 and 0, Ri	And with long immediate. The immediate value is made by inserting a pfx instruction.
Memory		
add Ri, [An, 32]	pfx 1 mov Ri, [An, 0]	Add Ri to memory location An+32. The immediate displacement does not fit in 5 bit, so a pfx instruction is inserted.
Zero Page		
add Ri, [K]	pfx K >> 5 add Ri, [K & 0x1f]	Zero page arithmetic. Destination address is beyond the 5 bit field, so a pfx instruction is inserted.
Relative Branch		
bt+ Offset	pfx Offset >> 5 b+ Offset & 0x1f	Conditional relative branch forward. PC displacement does not fit in immediate, so a pfx instruction is inserted.
Jump with Link		
jl K	pfx K >> 5 jl K & 0x1f	Jump with link. Address does not fit in immediate, so a pfx instruction is inserted.

The prefix instruction is available for the addressing modes specifying immediate fields, M-Type, ZP-Type, and I-Type. It must directly precede the affected instruction.

Arithmetic operations

The table below summarises all arithmetic instructions by mnemonic. More information on specific instructions is available on the following sections.

Addition:

Mnemonic	Description (*)
add	Add Source1 with Source2 and store in Destination. Update flags
dad	BCD add, Source1 with Source2 and store in Destination. Update flags
adc	Add with carry Source1 with Source2 and store in Destination. Update flags
dac	BCD add with carry Source1 with Source2 and store in Destination. Update flags
(*) These instructions are available for all addressing modes	

Subtraction:

Mnemonic	Description (*)
sub	Subtract Source2 from Source1, store in Destination, update flags (1)
rsb	Reverse Subtraction, subtract Source1 from Source2, store in Destination, update flags (1)
sbc	Subtraction with borrow (borrow == not carry), update flags
dsb	BCD subtraction, update flags (1)
dsc	BCD subtraction with borrow (borrow == not carry), update flags (1)
(*) These instructions are available for all addressing modes except M-Type (1) For I-Type mode, and ZP-Type loads, 'sub', 'sbc', 'dsc' subtract the first operand from the second operand, 'rsb' subtract the second operand from the first operand. For ZP-Type stores, 'sub', 'sbc', 'dsc' subtract the second operand from the first operand, 'rsb' subtract the second operand from the first operand. In all cases, the result is stored on the second operand. This is consistent with operand execution order described in the Addressing Modes section.	

Arithmetic instructions set C and Z condition flags according to the result. Additionally, the Z flag is copied to T for binary instructions and the C flag is copied to T for BCD instructions. For example, the 'add' instruction will set C to 1 if there was a carry, and both Z and T to 1 if the result was zero. The 'dad' instruction will set Z to 0 if the result was zero, and both C and T to 1 if there was a carry.

Bitwise operations

The table below summarises all arithmetic instructions by mnemonic. More information on specific instructions is available on the following sections.

Mnemonic	Description (*)
and	Bitwise and, update flags
or	Bitwise or, update flags
xor	Bitwise xor, update flags
(*) These instructions are available for all addressing modes except M-Type	

Bitwise instructions always clear the C flag, therefore the following instruction 'or 0, Ri' can be used for this effect alone. Both Z and T are set to true if the result was zero.

Carry-in instructions

A number of instructions take the carry flag to enable wider than native operations. For example, a 32 bit addition can be performed on two pairs of registers representing 32 bit values, by sequentially executing 'add' on the lower register operands or memory locations, followed by an 'adc' on the upper operands.

The following carry-in instructions are available:

Mnemonic	Description (*)
adc	Add with carry
dac	Decimal add with carry
sbc	Subtract with carry (1)
dsc	Decimal subtract with carry (1)
cpc	Compare with carry (2)

(1) See notes on Arithmetic Operations section
(2) See notes on Comparison Instructions section

Carry-in instructions are designed to be executed in combination with carry setting instructions of the same family. The Status Register flags after carry-in instructions will correctly reflect the result of the combined operation. Therefore it is safe to use conditional branch or move instructions after them.

Shift and Rotate instructions

Shift instructions perform logical shifts of 1 bit or 4 bit amounts on Source1 and store the result on Destination. Source2 is ignored. The shifted in bits are zeros.

Rotate instructions perform a shift on Source1 by using Source2 to shift in the required lower or upper bits to complete the shift. Combining rotate and shift instructions enable multi-word shifts.

Shift and rotate instructions:

Instruction	
sr1 Rj, Ri	1 bit shift right Rj, zero is shifted in, store in Ri
sr4 Rj, Ri	4 bits shift right Rj, zeros is shifted in, store in Ri
sl1 Rj, Ri	1 bit shift left Rj, 1 bit, zero is shifted in, store in Ri
sl4 Rj, Ri	4 bits shift left Rj, 4 bits, zeros is shifted in, store in Ri
rr1 Rj, Rk, Ri	1 bit rotate right Rj, 1 bit, shifted in bit taken from Rk, store in Ri
rr4 Rj, Rk, Ri	4 bits rotate right Rj, 4 bits, shifted in bits taken from Rk, store in Ri
rl1 Rj, Rk, Ri	1 bit rotate left Rj, 1 bit, shifted in bit taken from Rk, store in Ri
rl4 Rj, Rk, Ri	4 bits rotate left Rj, 4 bits, shifted in bits taken from Rk, store in Ri
Shifts or Rotates are performed on the first operand, result is stored on the last operand. Both C flag and T flags are set if the shifted out bit of a 1-bit shift is 1, or the shifted out nibble of a 4-bit shift is not Zero. Cleared otherwise. The same instructions exist for the ZP-Type and I-Type addressing modes except that Rj is replaced by Source1, Rk by Source2 and Ri by the Destination. See the Addressing Modes section for more information	

Example 1:

1 bit shift right of the contents of the 32 bit register pair R0:R1. Register R0 contains the least significative half:

```
rr1 r0, r1, r0  // shift r0 right by incorporating the lowest bit of r1 into the highest bit of r0
sr1 r1, r1      // shift r1 right
```

Example 2:

1 bit shift left of the contents of the 32 bit register pair R0:R1. Register R0 contains the least significative half:

```
rl1 r1, r0, r1  // shift r1 left by incorporating the highest bit of r0 into the lowest bit of r1
sl1 r0, r0      // shift r0 left
```

Example 3:

4 bit shift right of the contents of the 32 register pair R0:R1. Register R0 contains the least significative half:

```
rr4 r0, r1, r0  // shift r0 right by incorporating the lowest nibble of r1 into the highest nibble of r0
sr4 r1, r1      // shift r1 right by 4
```

Example 4:

4 bit shift left of the contents of the 32 bit register pair R0:R1. Register R0 contains the least significative half:

```
rl4 r1, r0, r1  // shift r1 left by incorporating the highest nibble of r0 into the lowest nibble of r1
sl4 r0, r0      // shift r0 left by 4
```

Example 5:

I-Type 4 bit rotates

```
rl4 0x7000, r0  // shift 0x7 into the lower nibble of R0
sl4 7, r0       // shift 0x7 into the upper nibble of R0
```

Shift and Rotate instructions (V2)

Shift instructions perform logical shifts of 1 bit or 4 bit amounts on Source2 and store the result on Destination. Source1 is ignored. Shifted in bits are all zeros. The shifted out bits are stored on the RR field of the Status Register.

Rotate instructions perform a shift on Source2 by shifting in the bits stored in the Status Register. Combining rotate and shift instructions enable multi-word shifts.

Shift and rotate instructions:

Instruction	
`sr1 Rk, Ri`	1 bit shift right Rk, zero is shifted in, store in Ri. Update SR
`sr4 Rk, Ri`	4 bits shift right Rk, zeros is shifted in, store in Ri. Update SR
`sl1 Rk, Ri`	1 bit shift left Rk, 1 bit, zero is shifted in, store in Ri. Update SR
`sl4 Rk, Ri`	4 bits shift left Rk, 4 bits, zeros is shifted in, store in Ri. Update SR
`rr1 Rk, Ri`	1 bit rotate right Rk, 1 bit, shifted in bit taken from SR, store in Ri
`rr4 Rk, Ri`	4 bits rotate right Rk, 4 bits, shifted in bits taken from SR, store in Ri
`rl1 Rk, Ri`	1 bit rotate left Rk, 1 bit, shifted in bit taken from SR, store in Ri
`rl4 Rk, Ri`	4 bits rotate left Rk, 4 bits, shifted in bits taken from SR, store in Ri
Both C flag and T flags are set if the shifted out bit of a 1-bit shift is 1, or the shifted out nibble of a 4-bit shift is not Zero. Cleared otherwise. All instructions update the RR field of the Status Register with the bits being shifted out. The Rotate instructions get the shifted in bits from the RR field of the SR The same instructions exist for the ZP-Type and I-Type addressing modes except that Rk is replaced by Source2, and Ri by the Destination. See the Addressing Modes section for more information	

Example 1:

1 bit shift right of the contents of the 32 bit register pair R0:R1. Register R0 contains the least significative half:

```
sr1 r1, r1      // shift r1 right
rr1 r0, r0      // shift r0 right by incorporating the lowest bit of r1 into the highest bit of r0
```

Example 2:

1 bit shift left of the contents of the 32 bit register pair R0:R1. Register R0 contains the least significative half:

```
sl1 r0, r0      // shift r0 left
rl1 r1, r1      // shift r1 left by incorporating the highest bit of r0 into the lowest bit of r1
```

Example 3:

4 bit shift right of the contents of the 32 register pair R0:R1. Register R0 contains the least significative half:

```
sr4 r1, r1      // shift r1 right by 4
rr4 r0, r0      // shift r0 right by incorporating the lowest nibble of r1 into the highest nibble of r0
```

Example 4:

4 bit shift left of the contents of the 32 bit register pair R0:R1. Register R0 contains the least significative half:

```
sl4 r0, r0      // shift r0 left by 4
rl4 r1, r1      // shift r1 left by incorporating the highest nibble of r0 into the lowest nibble of r1
```

Example 5:

I-Type shift

```
sl4 0x7, r0     // shift 0x7 4 bits left, so 0x70, and store in R0
sr4 0x5, r0     // shift 0x7 4 bits right and store in R0 (R0 now contains 0, and RR is 0x5)
rl4 0x7, r0     // shift 0x7 4 bits left while incorporating the contents of RR (R0 now contains 0x75)
```

Condition Codes

Encoding	Machine Name	Alt Names	SR Flags	Description
000	eq	z	Z	Equal than. Zero
001	ne	nz	!Z	Not equal. Not zero
010	uge	hs, c	C	Unsigned greater than or equal. Carry
011	ult	lo, nc	!C	Unsigned less than. Not carry
100	ge	-	S == V	Signed greater than or equal
101	lt	-	S != V	Signed less than
110	ugt	hi	C && !Z	Unsigned greater than
111	gt	-	(S == V) && !Z	Signed greater than
-	ule	ls	!C || Z	Unsigned less than or equal Implemented as the opposite of ugt
-	le	-	(S != V) || Z	Signed less than or equal Implemented as the opposite of gt

The S and V flags may be computed internally to match condition codes, but they are not stored in the SR register, nor are available to the user.

Comparisons

Comparisons are performed with the 'cmp' and 'cpc' instructions. The comparison instructions take two operands and a condition code to set the 'T' condition flag if the condition was met. The comparison is made by performing a binary subtraction. The processor supports both signed and unsigned comparisons. Wider than word comparisons can be carried out with the carry-in comparison instructions. A 32 bit comparison can be performed on pairs of operands representing 32 bit values, by sequentially executing 'cmp' on the lower pair followed by a 'cpc' on the upper pair. The condition flags after a 'cpc' preceded by a 'cmp' are guaranteed to be correct for the 32 bit comparison.

The following comparison instructions are available:

cmp [An, K], Ri cmp [K], Ri cmp.CC Rj, Rk cmp K, Ri	Compare the two operands by subtracting them. Update 'T' flag according to the given 'CC' condition code and comparison result. (1)
cpc [K], Ri cpc.CC Rj, Rk cpc K, Ri	Compare operands as above but take into account previous 'C' and 'Z' flags to appropriately. Update the 'T' flag. (1)

(1) The cmp instruction is available for all addressing modes, The cpc instruction is available for ZP-Type, R-Type and I-Type addressing modes.

For M-Type and ZP-Type instructions, only 'eq' is available. In this case the condition code is ommited

Example 1:

Compare the 32 bit register pair R0:R1 with R2:R3 for the less-than condition .

```
cmp.lt r0, r2    // test whether r0 < r2, these are the less significative pair
cpc.lt r1, r3    // text whether r1 < r3, with taking into account the previous comparison result.
```

Example 2:

Compare the 32 bit memory contents pointed to by register A0, with memory contents at address M

```
mov [a0, 0], r0  // load lower half of first value
cmp [&foo], r0   // compare with lower half of second value
mov [a0, 1], r0  // load higher half of first value
cpc [&bar], r0   // compare with higher half of second value, T flag is set if both 32 values are equal
```

Move instruction

The 'mov' instruction copies 16 bit data from one source to the destination operand. The source operand is not affected. Source1 or Source2 is chosen depending on whether it's a load or a store instruction as specified on the table below:

Mnemonic	Description (*)
mov	Copy the left operand to the right operand
(*) This instruction is available for all addressing modes, including loads and stores. For M-Type and ZP-Type loads, Source2 is copied to Destination. For stores, Source1 is copied to Destination. See the Addressing Modes section for more. The status register flags are not affected	

Load from Program Memory

Program Memory is accessed with the 'lp' instruction.

lp [Rk], Ri	Load contents of program memory address Rk into register Ri
This is a special instruction with no support for different addressing modes	

Conditional Moves and Selects

A number of instructions enable conditional copy or selects based on the 'T' condition flag. The following conditional instructions are available:

Conditional moves and selects:

set Rk, Ri	If 'T' is set, copy Rk to Ri, else set Ri to 0. {Ri = (T ? Rk : 0)}
sef Rk, Ri	If 'T' is not set, copy Rk to Ri, else set Ri to 0. {Ri = (T ? 0 : Rk)}
sel Rj, Rk, Ri	If 'T' is set, copy Rk to Ri, else copy Rj to Ri. {Ri = (T ? Rk : Rj)}
The same instructions exist for ZP-Type and I-Type addressing mode except that Rj is replaced by Source1, Rk by Source2 and Ri by the Destination. See the Addressing Modes section for more information	

Faster execution by prevention of branching code can be achieved in simple cases by conditional moves. For example, the 'booth' multiplication algorithm requires an 'add' instruction to be skipped based on the multiplicand term. This can be implemented by placing a 'set' instruction conditionally reseting a temporary value before the addition is executed. (See program examples on the Assembly Instruction Format section)

Note that a Conditional Move if 'T' is set instruction can be achieved with the 'sel' instruction when the destination operand is the same as the first operand. Similarly, a Conditional Move if 'T' is not set instruction can be achieved with the 'sel' instruction when the destination operand is the same as the second operand.

Example 1:

Execute a Conditional Move instruction when 'T' is set

```
sel R0, R1, R0   // If 'T', copy R1 to R0, else copy R0 to R0. Therefore, move R1 to R0 if 'T' is set
```

Example 2:

Execute a Conditional Move instruction when 'T' is not set

```
sel R1, R0, R0   // If 'T', copy R0 to R0, else copy R1 to R0. Therefore, move R1 to R0 if 'T' is not set
```

Conditional Arithmetic

The following instructions enable conditional execution of binary arithmetic based on the 'T' condition flag.

Mnemonic	Description (*)
adt	Binary add, store result in destination only if T flag was previously set
adf	Binary add, store result in destination only if T flag was previously cleared
sbt	Binary subtract, store result in destination only if T flag was previously set
sbf	Binary subtract, store result in destination only if T flag was previously cleared
(*) These instructions are available for ZP-Type, R-Type and I-Type addressing modes, including loads and stores. No Status register flags are updated as a result of these instructions	

Faster execution by prevention of branching code can be achieved in simple cases by conditional arithmetic. For example, the 'booth' multiplication algorithm requires an 'add' instruction to be skipped based on the multiplicand term. This can be implemented by placing a 'set' instruction conditionally reseting a temporary value before the addition is executed.

Multiply using the 'booth' algorithm. End when the multiplicand is zero. The core multiplication uses up to 97 cycles but will be much faster for small multiplicands

```
         mov 100, a0          // assume 100 is the top of the stack address
         mov [a0, 0], r1      // get multiplier
         mov [a0, 1], r2      // get multiplicand
         mov 0, r0            // set result to zero
.LMulHi
         cmp.eq 0, r2         // compare multiplicand with zero
         bt+ .LMulDone        // branch if zero
         sr1 r2, r2           // shift right the multiplicand
         adt r0, r1, r0       // add multiplier (or nothing)
         sl1 r1, r1           // shift multiplier left
         b- .LMulHi           // next iteration
.LMulDone
         add 1, a0            // increment the data stack pointer
         mov r0, [a0, 0]      // store the result on top of the stack
```

<u>Program example 2</u>

Multiply using the 'booth' algorithm. Constant execution time. The core multiplication uses 80 cycles

```
         mov 100, a0          // assume 100 is the top of the stack address
         mov [a0, 0], r1      // get multiplier
         mov [a0, 1], r2      // get multiplicand
         add 1, a0            // increment stack address
         mov 0, r0
         mov r0, [a0, 0]      // set initial result to zero
         mov 16, r0           // initialise counter
.LMulHi
         sr1 r2, r2           // shift right the multiplicand
         adt r1, [a0, 0]      // conditionally accumulate the result
         sl1 r1, r1           // shift multiplier left
         sub 1, r0            // decrement counter
         bt- .LMulHi          // next iteration
.LMulDone
         // done, the stack pointer is already incremented
         // and the result in the right memory location
```

Branch and Jump

All instructions operating on registers can be normally used with the Program Counter (PC) as with any other register. However, the assembler conveniently renames the I-Type instructions that use the PC as the destination operand as jumps or branches. The underlying instruction encodings are identical to the equivalent I-Type instructions:

Instruction	Equivalent	
`j .Label`	`mov Label, PC`	Absolute Jump, unconditional
`b+ .Label`	`add Label, PC`	PC relative branch forward, unconditional (1)
`b- .Label`	`sub Label, PC`	PC relative branch backward, unconditional (1)
`bt+ .Label`	`adt Label, PC`	PC relative branch forward, if 'T' flag is set
`bf+ .Label`	`adf Label, PC`	PC relative branch forward, if 'T' flag is not set
`bt- .Label`	`sbt Label, PC`	PC relative branch backward, if 'T' flag is set
`bf- .Label`	`sbf Label, PC`	PC relative branch backward, if 'T' flag is not set
(1) It's important to consider that, as a side effect, the 'b+' and 'b-' instructions will update the SR flags as any 'add', 'sub' instruction would do. Use the 'j' instruction to prevent this effect.		

Note that instructions involving jumps are not limited to these ones, as the PC is just one more register. It is possible for example to implement jump tables by using R-Type, ZP-Type or M-Type instructions with the PC as the destination.

Move and Link, Jump and Link, Subroutines

The 'mvl' instruction is a special 'mov' instruction that will store the address of the following instruction into register R6 (or A3) before executing the move.

Mnemonic	Description (*)
mvl	Store PC+1 in R6, then execute as a normal 'mov'
(*) This instruction is available for ZP-Type, R-Type and I-Type addressing modes, including loads and stores. The Status Register flags are unafected	

When used with the PC as the destination (`'mvl K, PC'`) the instruction gets the name of Jump and Link (`'jl Label'`)

Instruction	Equivalent	
`jl Label`	`mvl Label, PC`	Absolute Jump and Link

The 'jl' instruction provides a way to implement subroutine calls. The instruction performs as a normal absolute jump, but it will copy PC+1 to register R6 (or A3) before executing the jump. This enables the callee to return to the caller address by simply executing `'mov r6, PC'`.

As noted on the Branch and Jump section, instructions involving jumps are not limited to I-Type ones, as the PC can be used as the destination on all addressing modes. It is possible for example to implement subroutine call tables by using R-Type, ZP-Type or M-Type instructions.

Stack based call frames are not natively supported but they can be implemented by storing the return address on the stack just a the beginning of the called subroutine.

<u>Program example1</u>

Call to subroutine (simple version)

```
AddSubroutine:
         add r0, r1, r0       // perform addition
         mov r6, PC           // return to caller

ProgramEntry:
         mov 100, r0          // set r0 to 100
         mov 3, r1            // set r1 to 2
         jl AddSubroutine     // call AddSubroutine to perform addition
         mov r0, [&Result]    // store the result in memory
```

```
        hlt                  // stop execution

        .comm result,2,2     // reserve space in data memory for result
```

Program example 2

Call to subroutine (fixed stack frame version)

```
AddSubroutine:
        mov r6, [&addSubReturn]   // store the return address
        add r0, r1, r0            // perform addition
        mov [&addSubReturn], PC   // return to caller

ProgramEntry:
        mov 100, r0               // set r0 to 100
        mov 3, r1                 // set r1 to 2
        jl AddSubroutine          // call AddSubroutine to perform addition
        mov r0, [&result]         // store the result in memory
        hlt                       // stop execution

        .comm adSubReturn,2,2     // reserve space in data memory for return address
        .comm result,2,2          // reserve space in data memory for result
```

Program example 3

Call to subroutine (dynamic stack frame version, r5 is used as the stack pointer)

```
AddSubroutine:
        sub 32, r5                // reserve 32 word frame space
        mov r6, [r5, 31]          // save return address,
                                  // [r5,0] to [r5,30] are now available for local vars
        add r0, r1, r0            // perform addition
        add 32, r5                // restore the caller stack frame
        mov [r5, 1], PC           // return to caller

ProgramEntry:
        mov [&stackData+1024], r5     // set r5 to the top of the stack, r5 becomes the stack pointer
        mov 100, r0               // set r0 to 100
        mov 3, r1                 // set r1 to 2
        jl AddSubroutine          // call AddSubroutine to perform addition
        mov r0, [&result]         // store the result in memory
        hlt                       // stop execution

        .data
stack:  .short stackData          // initialise stack with the bottom address of the reserved stack area
        .comm result,2,2          // reserve space in data memory for result
        .comm stackData, 1024, 64 // reserve 1024 bytes of 64-byte aligned data space for stack storage
```

Decimal Accumulation Instructions

Two special instructions are provided to speed up the generation of product multiple tables that must be created for decimal multiplications.

r0a Ri, [a0, K]	Decimal add of R0 with Ri. Store the result in R0, and in [a0, K]
r1a Ri, [a0, K]	Decimal add with carry of R1 with Ri. Store the result in R1, and in [a0, K]
This is a special instruction with no support for different addressing modes.	

The r0a, and r10 instructions have the following equivalences, but they only take one instruction cycle to execute

Instruction	Equivalent

r0a Ri, [a0, K]	`dad ri, r0, r0    // decimal add`
	`mov r0, [a0, K]   // M-Type store`

r1a Ri, [a0, K]	`dac ri, r1, r1    // decimal add`
	`mov r1, [a0, K]   // M-Type store`

No-operation Instruction

The 'nop' instruction advances the processor Program Counter without performing any operation. It is emulated through the 'pfix' instruction with all bits set to zero. pclock and waits for a key to be pressed to continue execution

Mnemonic	Equivalent	
nop	pfx 0	No Operation

Halt Instruction

The 'hlt' instructions stops the processor clock and waits for a key to be pressed to continue execution

hlt	Halt processor
(*) This is a special instruction with no support for different addressing modes	