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 Hardvard 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.

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			Ak		s	immediate (5 bits)					Indexed Memory Addressing
ZP	01		op			Ri			fn		s	immediate (5 bits)					Direct Memory Addressing
R	10		op			Ri/CC			fn		Rj			Rk			Three registers
I	11		op			Ri			fn		0	immediate (5 bits)					Immediate, Branch
P	11		000			immediate (11 bits)											Prefix

op: 3 bit opcode for the instruction type
fn: 2 bit function code
s : 1 bit field indicating that the instruction is a memory store

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
PC	A3	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 refer to program memory rather than data memory.

P is the prefix register. It's written by the prefix instruction and implicitly used by type I instructions.

There's no explicit 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
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 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.	

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 one 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 source operands, operand 3 is the destination.

* M-Type and ZP-Type instructions take 2 operands, a register operand 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 immeditate 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
        sef 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
```

Program example 2

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
         sef 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
```

Addressing Modes

The following table summarises the available addressing modes. Addressing modes relate with instruction Types

Type	Addressing mode	Source 1	Source 2	Destination	Example
I	Immediate	(1)	K	Ri	add 4, r0
	Branch			PC	b+ 4 (same as 'add 4, PC')
M	Indexed Memory load	(1)	mem(Aj+K)	Ri	add [a0, 4], r0
	Indexed Memory store	(1)	Ri	mem(Aj+K)	add r0, [a0, 4]
R	Register (2) (3)	Rj	Rk	Ri	add r1, r2, r0
ZP	Direct Memory load	(1)	mem(K)	Ri	add [M], r0
	Direct Memory store	(1)	Ri	mem(K)	add r0, [M]

(1) Same as destination.

(2) The 'set', 'sef', 'sl1', 'sl4', 'sr1', 'sr4', instructions belonging to the R-Type slightly modify the default R addressing mode by ignoring one of the source operands.

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

Instruction Encodings

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

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.

Prefixed instructions

Prefixed instructions are assembler emulated instructions of type M, ZP, or I that are made of core instructions preceded by a prefix instruction. The prefix instruction contains a 'p_imm' 11 bit immediate field that expands the functionality of core instructions. The prefix instruction extends the immediate field 'imm' of the next instruction by replacing it with the result of the logical expression: (p_imm << 5) | (imm & 0b11111), thus providing 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, [Aj, 32]	pfx 1 mov Ri, [Aj, 0]	Add Ri to memory location Aj+32. The immediate displacement does not fit in 5 bit, so a pfx instruction is inserted.
Zero Page		
add Ri, [M]	pfx M >> 5 add Ri, [M & 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.
Branch with Link		
brl M	pfx M >> 5 brl M & 0x1f	Branch with link. Address does not fit in immediate, so a pfx instruction is inserted.

Arithmetic operations

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

Mnemonic	Description (*)
add	Binary add, update flags
dad	BCD add, update flags
adc	Binary add with carry, update flags
dac	BCD add with carry, update flags
sub	Subtraction, update flags (1)
rsb	Reverse Subtraction, update flags (1)
sbc	Subtraction with borrow (borrow == not carry), update flags
dsc	BCD subtraction with borrow (borrow == not carry), update flags (1)(2)
(*) All addressing modes including loads and stores are available (1) For I-Type, M-Type 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 M-Type and 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. (2) There is no BCD subtract without carry. Carry flag must be cleared before if required.	

Arithmetic instructions set C and Z condition flags according to the result. Additionally, the Z 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.

Bitwise operations

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

Mnemonic	Description (*)
and	Bitwise and, update flags
or	Bitwise or, update flags
xor	Bitwise xor, update flags
(*) All addressing modes including loads and stores are available	

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.

Move instructions

The table below summarises the move instructions on specific instructions is available on the following sections.

Mnemonic	Description (*)
mov	Copy the left operand to the right operand
(*) All addressing modes including loads and stores are available	

Move instructions do not affect Status Flags

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:

adc [M], Ri adc Ri, [M] adc Rj, Rk, Ri adc K, Ri	Add with carry
dac [M], Ri dac Ri, [M] dac Rj, Rk, Ri dac K, Ri	Decimal add with carry
sbc [M], Ri sbc Ri, [M] sbc Rj, Rk, Ri sbc K, Ri	Subtract with carry (1)
dsc [M], Ri dsc Ri, [M] dsc Rj, Rk, Ri dsc K, Ri	Decimal subtract with carry (1)
cpc.eq [M], Ri cpc.eq Ri, [M] cpc.<cc> Rj, Rk, cpc.eq K, Ri	Compare with carry
(1) See notes on Arithmetic Operations 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 and 4 bit shift amounts on register operands. Rotate instructions take two register operands and use the second operand to shift in the required lower or upper bits into the first operand to complete the shift. By carefully combining shift and rotate instructions any shift amount on multi-word data is possible.

`sr1 Rj, Ri`	Logical shift right, 1 bit Rj and stores in Ri
`sr4 Rj, Ri`	Logical shift right, 4 bits
`sl1 Rj, Ri`	Logical shift left, 1 bit
`sl4 Rj, Ri`	Logical shift left, 4 bits
`rr1 Rj, Rk, Ri`	Rotate right, 1 bit, Rk contains the incoming bits
`rr4 Rj, Rk, Ri`	Rotate shift right, 4 bits, Rk contains the incoming bits
`rl1 Rj, Rk, Ri`	Rotate shift left, 1 bit, Rk contains the incoming bits
`rl4 Rj, Rk, Ri`	Rotate shift left, 4 bits, Rk contains the incoming bits
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.	

<u>Example 1</u>:

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
```

<u>Example 2</u>:

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
```

<u>Example 3</u>:

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
```

<u>Example 4</u>:

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
```

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	ule	ls	!C || Z	Unsigned less than or equal
111	le	-	(S != V) || Z	Signed less than or equal
-	ugt	hi	C && !Z	Unsigned greater than Implemented as the opposite of ule
-	gt	-	(S == V) && !Z	Signed greater than Implemented as the opposite of le

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 matched after comparing the two operands. 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 two pairs of operands representing 32 bit values, by sequentially executing 'cmp' on the lower operand 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.<cc> Rj, Rk cmp.eq [M], Ri cmp.eq [Aj, K], Ri	Compare the two operands by subtracting them. Update 'T' flag according to the given 'cc' condition code and comparison result. (1)
cpc.cc Rj, Rk cpc.eq [M], Ri cpc.eq [Aj, K], Ri	Compare operands as above but take into account previous 'C' and 'Z' flags to appropriately. Update the 'T' flag.

(1) Only 'eq' is allowed for M-Type and ZP-Type instructions.

<u>Example 1</u>:

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.
```

<u>Example 2</u>:

Compare the 32 bit memory contents pointed to by register A0, with memory contents pointed to by register A1 for equality

```
mov [a0, 0], r0      // load lower half of first value
cmp.eq [a1, 0], r0   // compare with lower half of second value
mov [a0, 1], r0      // load higher half of first value
cpc.eq [a1, 1], r0   // compare with higher half of second value
```

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:

`set Rk, Ri`	`If 'T' is set, copy Rk to Ri, else set Ri to 0.     {Ri = (T ? Rj : 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 ? Rj : Rk)}`
`The same rules apply for M-Type and ZP-Type instructions except that Rj is the same as the destination operand, and Rk is replaced by the corresponding memory addressing mode.`	

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.

<u>Example 1</u>:

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
```

<u>Example 2</u>:

Execute a Conditional Move instruction when 'T' is not 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
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
(*) All addressing modes including loads and stores are available. 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
         adf 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
         adf 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
```

Branches

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

Mnemonic	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'
(*) All addressing modes including loads and stores are available	

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

Mnemonic	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, 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.