2022

GS-CPU12

GAMESEED 12-BIT CPU ARCHITECTURE

BEEFOK, @BEEFOK, GAMESEED-GX@PROTONMAIL.COM

TABLE OF CONTENTS

PROGRAMMER'S MODEL

Every instruction involves 0 to 2 general purpose registers. These registers are labeled R0-R7 and have been allocated in the programmer's model this way:

NAME	ALIAS	DESCRIPTION
R0	ZR	Always zero value
R1	AS	Assembler Use / Return Address
R2	AX	Subroutine Return Value / General Purpose
R3	BX	Local Variable / General Purpose
R4	CX	Local Variable / General Purpose
R5	DX	Local Variable / General Purpose
R6	BP	Base (Frame) Pointer
R7	SP	Stack Pointer

All registers can be used however you want, but the 'best use' organization is summarized above. Outside of the general purpose registers used in instructions, there are also internal instructions that are operated on in other ways.

NAME	DESCRIPTION
PC	Program Counter, location of the current instruction to be executed
RA	Return Address, location of next instruction to be executed in User Mode
IR	Interrupt Register, current pending interrupts, masks, and enables.
IW	Instruction Word, holds the currently decoded instruction, and also holds onto the next instruction to be executed during a stall/memory access operation.

These registers are handled by the CPU itself. The **program counter** is operated on with the flow control instructions. The **return address** is updated whenever the CPU jumps to machine mode, which allows the machine to return back to user mode after it is done. The interrupt register holds the current status of external interrupts, as well as masks and CPU mode (0 = machine mode, 1 = user mode). This register can only be operated on by machine mode.

Invisible to the programmer, the CPU has a three-stage pipeline, separated into **Fetch**, **Decode**, and **Execute** stages.

In the **fetch stage**, the CPU requests access to the external bus. It can read and write to the bus and also outputs the current state of the CPU. The address bus is 12-bit, the data bus is 12-bit, and it also includes if it is in user-mode and if this is a code memory request vs a data memory request.

In the **decode stage**, the CPU decodes the currently accessed bus word as an instruction, or, depending on the current state of the CPU, whether it requests a memory word from the bus. If the CPU is in user mode, and there is an external interrupt request, a BRK instruction is decoded instead of the next instruction to be decoded. The decode stage also handles state changes such as user mode switch, interrupt masking, user return address updating.

In the **execute stage**, the CPU executes the instruction that has been presented to it by the **decode stage**. It also updates the bus actions to be used in the **fetch stage**. In this stage, the **register file** reads out two registers (register X and register Y), and whatever operation needs to be handled happens and the result is written back to either the register file, the program counter, or to the bus. The **execute stage** is also responsible for updating the **program counter**.

Most instructions are executed in a single cycle, however, memory operations take three cycles to complete, and the control flow instructions take two cycles. ALU instructions take two cycles if they use a 12-bit operand in the place of register Y, and one cycle if they use register Y or a small immediate as the operand.

NAME	DESCRIPTION	clocks / instruction
LW/SW	Memory access	3
BRK/JZ/JR	Control Flow	2
ALU, 12-bit IMM	ALU operation	2
ALU, RY or 3-bit IMM	ALU operation	1

INSTRUCTION SET LISTING

The CPU has 18 instructions with various addressing modes for most instructions. Each set of instructions will be organized by their use.

This is the format of an entry:

`NAME <OPERANDS, …>`

`Instruction word: (0000 0000 0000)`

<Description of instruction>

Uses: <Where can it be used>

Action:

```
<Description of actions taken by CPU when executing this instruction>
```

If the instruction word requires two words, it is represented as: `(0000 0000 0000 IIII IIII IIII)`

MACHINE AND USER STATE CONTROL

BRK IMM3

Instruction word: (0XXX 0000 0000)

In user mode this instruction is known as BRK (break from user mode), which is the system call and interrupt handling mechanism. On entry of machine mode, all external interrupts are masked and cannot be unmasked. In order to handle external interrupts, machine mode must jump back to user mode as quickly as possible.

An external interrupt request pushes a BRK instruction into the pipeline preventing a new instruction from being placed into the pipeline. On entry of an interrupt, the user code address that would have been executed is placed into the User Program Counter register (known as UPC from now on.). In machine mode, the BRK instruction can then be used to return back to user mode.

There are 8 total interrupts that can be handled. These interrupts are handled with decreasing priority where interrupt 0 is highest priority compared to interrupt 7, which is the lowest priority.

Each interrupt has 8 words allocated to it in machine mode code space.

Interrupt ID	description	address: begin	end
irq 0 (reset)	reset (non-maskable)	0x000	0x007
irq 1 (external)	typically for external interrupts	0x008	0x00F
irq 2		0x010	0x017
irq 3		0x018	0x01F
irq 4		0x020	0x027
irq 5		0x028	0x02F
irq 6		0x030	0x037
irq 7 (usr reqs)	user requests	0x038	0x03F+

Typically, these interrupt handlers will be used to jump to a specific subroutine allocated for it further in the machine mode code space. However, they also have enough space to store a few important registers such as the stack pointer and frame pointer.

Uses: system reset, handle an external event, user system call, force/spoof external event from user mode.

Action:

```
if USER: PC = IRQ.X, UPC = PC, USER = 0
```

`REU (BRK R0)`

`Instruction word: (0000 0000 0000)`

To return to user mode, machine mode needs to execute a BRK instruction. In machine mode, the BRK instruction has another name, known either as JRU (jump to register with user address) or REU (return to user mode with UPC). It can still be called BRK, but for verbosity and the fact that the source X parameter is handled differently, it makes it easier to understand.

Uses: Return to user mode

Action:

```
if not USER: PC = UPC, USER = 1
```

`JRU RX`

`Instruction word: (0XXX 0000 0000)`

If the RX field is r0, it will use UPC to jump back to where the next user code address would have been fetched and executed. Once back in user mode, external interrupts can be requested and executed again. If the RX field is not r0, and is instead r1-r7, it will use the value in the register as the user code address to jump to. Once back in user mode, external interrupts can be requested and executed again.

Uses: Context switching, user mode reset, trusted user-handled interrupt/device driver.

`Instruction word: (0XXX 0000 0000)`

Action:

```
if not USER: PC = RX, USER = 1
```

`JZ RX, HIMM8`

`Instruction word: (0XXX HHHH HHHH)`

If the `HIMM immediate field` is not `00000000`, this is decoded as a jump on zero conditional branch instead of a BRK instruction. This is the only conditional instruction and can be combined with the comparison instructions in order to handle control flow. The takes the value stored in register X, compares it to zero, and jumps to the 8-bit PC-relative immediate, which gives jump range from -127 to +128 words relative to the next instruction. The reason why a JZ with an `HIMM immediate field` of `00000000` is a BRK instruction is because it would otherwise be decoded as a `NOP` instruction, which there are already many ways to achieve.

Action:

```
if RX == 0: PC = PC + SIMM8
```

MEMORY ACCESS

`LW RX, RY`

`Instruction word: (1XXX 1YYY 0000)`

Load a word from memory with the address calculated from the value stored in register Y and stores the word from memory into register X.

Access memory using register X, stack pop

Action:

```
RX = MEM[RY]
```

`LW RX, RY, IMM12`

`Instruction word: (1XXX 0YYY 0000 IIII IIII IIII)`

Loads a word from memory with the address calculated from the following instruction word decoded as a 12-bit immediate added to value stored in register Y and stores the world from memory into register X.

Action:

```
RX = MEM[RY + IMM12]
```

`SW RX, RY`

`Instruction word: (1XXX 1YYY 0001)`

Stores the word from register X into memory with the address calculated from the stored value in register Y.

Action:

```
MEM[RY] = RX
```

`SW RX, RY, IMM12`

`Instruction word: (1XXX 0YYY 0001 IIII IIII IIII)`

Stores the word from register X into memory with the address calculated from the following instruction word decoded as a 12-bit immediate added to value stored in register Y.

Action:

```
MEM[RY + IMM12] = RX
```

SUBROUTINE/JUMP TABLE FLOW CONTROL

JR RX, RY

`Instruction word: (1XXX 1YYY 0010)`

Jump to the address calculated from the value stored in register Y and store the instruction address after this instruction into register X. This instruction provides the system with the ability to jump to an indirect subroutine, and return from a subroutine.

Action:

```
PC = RY, RX = PC
```

JR RX, RY, IMM12

`Instruction word: (1XXX 0YYY 0010 IIII IIII IIII)`

Jump to the address calculated from the value stored in register Y added to the following instruction word decoded as a 12-bit immediate and store instruction address after the immediate instruction word into register X. This instruction provides the system with the ability to jump to subroutine, use jump tables, and to return from a subroutine.

Action:

```
PC = RY + IMM12, RX = PC
```

REGISTER TRANSFER

`MV     RX, RY`

`Instruction word: (1XXX 1YYY 0011)`

Store the value from register Y into register X.

Action:

```
RX = RY
```

`MV     RX, SIMM3`

`Instruction word: (1XXX 0YYY 0011)`

Store the small signed 3-bit immediate into register X.

Action:

```
RX = SIMM3
```

`MV     RX, IMM12`

`Instruction word: (1XXX 1YYY 0011 IIII IIII IIII)`

Store the following instruction word decoded as a 12-bit immediate into register X.

Action:

```
RX = IMM12
```

BITWISE SHIFT

LSL RX, RY

`Instruction word: (1XXX 1YYY 0100)`

Compute the logical shift left with the values stored in register X and Y, then store the result into register X. The lower 4-bits of register Y is the step amount is in range 0-15, where anything above 11 is just pure zero extension. The upper 8-bits of register Y are ignored.

Action:

```
RX = RX LSL RY
```

LSL RX, UIMM3

`Instruction word: (1XXX 0YYY 0100)`

Compute the logical shift right with the values stored in register X and the ‘shift’ immediate Y, then store the result into register X. The shift immediate value from 1-7 are 1 to 7-bit shift amounts, shift immediate 0 is an 8-bit shift amount.

Action:

```
RX = RX LSL BIMM3
```

LSR RX, RY

`Instruction word: (1XXX 1YYY 0101)`

Compute the logical shift right with the values stored in register X and Y, then store the result into register X. The lower 4-bits of register Y is the step amount is in range 0-15, where anything above 11 is just pure zero extension. The upper 8-bits of register Y are ignored.

Action:

```
RX = RX LSR RY
```

LSR RX, UIMM3

`Instruction word: (1XXX 0YYY 0101)`

Compute the logical shift right with the values stored in register X and the 'shift' immediate Y, then store the result into register X. The shift immediate value from 1-7 are 1 to 7-bit shift amounts, shift immediate 0 is an 8-bit shift amount.

Action:

```
RX = RX LSR BIMM3
```

`ASR   RX, RY`

`Instruction word: (1XXX 0YYY 0110)`

Compute the arithmetic shift right with the values stored in register X and Y, then store the result into register X. The lower 4-bits of register Y is the step amount is in range 0-15, where anything above 11 is just pure sign extension. The upper 8-bits of register Y are ignored.

Action:

```
RX = RX ASR RY
```

`ASR   RX, UIMM3`

`Instruction word: (1XXX 0YYY 0110)`

Compute the arithmetic shift right with the values stored in register X and the 'shift' immediate Y, then store the result into register X. The shift immediate value from 1-7 are 1 to 7-bit shift amounts, shift immediate 0 is an 8-bit shift amount.

Action:

```
RX = RX ASR BIMM3
```

COMPARISON

EQ RX, RY

`Instruction word: (1XXX 1YYY 0111)`

If the value stored in register X is equal to the value stored in register Y, store 1 into register X, otherwise store 0 into register X.

Action:

```
RX = RX == RY ? 0x001 : 0x000
```

EQ RX, SIMM3

`Instruction word: (1XXX 0YYY 0111)`

If the value stored in register X is equal to the small signed 3-bit immediate, then store 1 into register X, otherwise store 0 into register X.

Action:

```
RX = RX == SIMM3? 0x001 : 0x000
```

EQ RX, IMM12

`Instruction word: (1XXX 0YYY 0111 IIII IIII IIII)`

If the value stored in register X is equal to the following instruction word decoded as a 12-bit immediate, then store 1 into register X, otherwise store 0 into register X.

Action:

```
RX = RX == IMM12? 0x001 : 0x000
```

LR RX, RY

`Instruction word: (1XXX 1YYY 1000)`

If the value stored in register X is less than the value stored in register Y, store 1 into register X, otherwise store 0 into register X.

Action:

```
RX = RX < RY ? 0x001 : 0x000
```

`LR    RX, SIMM3`

`Instruction word: (1XXX 0YYY 1000)`

If the value stored in register X is less than the small signed 3-bit immediate, then store 1 into register X, otherwise store 0 into register X.

Action:

```
RX = RX < SIMM3? 0x001 : 0x000
```

`LR    RX, IMM12`

`Instruction word: (1XXX 0YYY 1000 IIII IIII IIII)`

If the value stored in register X is less than the following instruction word decoded as a 12-bit immediate, then store 1 into register X, otherwise store 0 into register X.

Action:

```
RX = RX < IMM12? 0x001 : 0x000
```

`GR    RX, RY`

`Instruction word: (1XXX 1YYY 1001)`

If the value stored in register X is greater than the value stored in register Y, store 1 into register X, otherwise store 0 into register X.

Action:

```
RX = RX > RY ? 0x001 : 0x000
```

`GR    RX, SIMM3`

`Instruction word: (1XXX 0YYY 1001)`

If the value stored in register X is greater than the small signed 3-bit immediate, then store 1 into register X, otherwise store 0 into register X.

Action:

```
RX = RX > SIMM3? 0x001 : 0x000
```

GR RX, IMM12

```
Instruction word: (1XXX 0YYY 1001 IIII IIII IIII)
```

If the value stored in register X is greater than the following instruction word decoded as a 12-bit immediate, then store 1 into register X, otherwise store 0 into register X.

Action:

```
RX = RX > IMM12? 0x001 : 0x000
```

ARITHMETIC

ADD RX, RY

```
Instruction word: (1XXX 1YYY 1010)
```

Compute the addition with the values stored in register X and Y, then store the result into register X.

Action:

```
RX = RX + RY
```

ADD RX, UIMM3

```
Instruction word: (1XXX 1YYY 1010)
```

Compute the addition with the value stored in register X and the small unsigned 3-bit immediate, then store the result into register X.

Action:

```
RX = RX - UIMM3
```

ADD RX, IMM12

```
Instruction word: (1XXX 1YYY 1010 IIII IIII IIII)
```

Compute the addition with the value stored in register X and the following instruction word decoded as a 12-bit immediate, then store the result into register X.

Action:

```
RX = RX + IMM12
```

`SUB   RX, RY`

`Instruction word: (1XXX 1YYY 1011)`

Compute the addition with the values stored in register X and Y, then store the result into register X.

Action:

```
RX = RX + RY
```

`SUB   RX, UIMM3`

`Instruction word: (1XXX 1YYY 1011)`

Compute the subtraction with the value stored in register X and the small unsigned 3-bit immediate, then store the result into register X.

Action:

```
RX = RX - UIMM3
```

`SUB   RX, IMM12`

`Instruction word: (1XXX 1YYY 1011 IIII IIII IIII)`

Compute the subtraction with the value stored in register X and the following instruction word decoded as a 12-bit immediate, then the result into register X.

Action:

```
RX = RX - IMM12
```

`RSB   RX, RY`

`Instruction word: (1XXX 1YYY 1100)`

Compute the ‘reverse’ subtraction with the values stored in register X and Y, then store the result into register X.

Action:

```
RX = RY - RX
```

`RSB   RX, UIMM3`

`Instruction word: (1XXX 1YYY 1100)`

Compute the 'reverse' subtraction with the value stored in register X and the small unsigned 3-bit immediate, then store the result into register X.

Action:

```
RX = UIMM3- RX
```

`RSB   RX, IMM12`

`Instruction word: (1XXX 1YYY 1100 IIII IIII IIII)`

Compute the 'reverse' subtraction with the value stored in register X and the following instruction word decoded as a 12-bit immediate, then store the result into register X.

Action:

```
RX = IMM12 - RX
```

BITWISE LOGIC

AND RX, RY

`Instruction word: (1XXX 1YYY 1101)`

Compute the bitwise-and with the values stored in register X and Y, then store the result into register X.

Action:

```
RX = RX AND RY
```

AND RX, SIMM3

`Instruction word: (1XXX 1YYY 1101)`

Compute the bitwise-and with the value stored in register X and the small signed 3-bit immediate, then store the result into register X.

Action:

```
RX = RX AND SIMM3
```

AND RX, IMM12

`Instruction word: (1XXX 1YYY 1101 IIII IIII IIII)`

Compute the bitwise-and with the value stored in register X and the following instruction word decoded as a 12-bit immediate, then store the result into register X.

Action:

```
RX = RX AND IMM12
```

OR RX, RY

`Instruction word: (1XXX 1YYY 1110)`

Compute the bitwise-or with the values stored in register X and Y, then store the result into register X.

Action:

```
RX = RX OR RY
```

`OR    RX, SIMM3`

`Instruction word: (1XXX 1YYY 1110)`

Compute the bitwise-or with the value stored in register X and the small signed 3-bit immediate, then store the result into register X.

Action:

```
RX = RX OR SIMM3
```

`OR    RX, IMM12`

`Instruction word: (1XXX 1YYY 1110IIII IIII IIII)`

Compute the bitwise-or with the value stored in register X and the following instruction word decoded as a 12-bit immediate, then store the result into register X.

Action:

```
RX = RX OR IMM12
```

`XOR   RX, RY`

`Instruction word: (1XXX 1YYY 1111)`

Compute the bitwise exclusive-or with the values stored in register X and Y, then store the result into register X.

Action:

```
RX = RX XOR RY
```

`XOR   RX, SIMM3`

`Instruction word: (1XXX 1YYY 1111)`

Compute the bitwise exclusive-or with the value stored in register X and the small signed 3-bit immediate, then store the result into register X.

Action:

```
RX = RX XOR SIMM3
```

XOR   RX, IMM12

`Instruction word: (1XXX 1YYY 1111 IIII IIII IIII)`

Compute the bitwise exclusive-or with the value stored in register X and the following instruction word decoded as a 12-bit immediate, then store the result into register X.

Action:

```
RX = RX XOR IMM12
```

EXAMPLE USAGE OF INSTRUCTIONS

Listed below are examples using CPU12 assembly as well pseudo-instruction definitions and usage.

STACK OPERATIONS

```
POP RX:                ; POP RX FROM STACK
  SW  RX, SP
  SUB SP, 1

PSH RX:                ; PUSH RX TO STACK
  ADD SP, 1
  LW  RX, SP

DUP:                   ; DUPLICATE TOP STACK ENTRY
  POP AS
  PSH AS
  PSH AS

SWAP:                  ; SWAP TWO TOP STACK ENTRIES
  POP AX
  POP BX
  PSH AX
  PSH BX
```

FUNCTION OPERATIONS

Function prologue and epilogue can be handled as follows:

```
ENTER #Y:
  PSH AS               ; ADD START OF FUNCTION, RETURN ADDRESS IS IN AS
  PSH BP               ; PUSH FRAME POINTER
  MV  BP, SP           ; SET FRAME POINTER TO NEW FRAME (STACK POINTER)
  SUB SP, #Y           ; MAKE ROOM FOR Y LOCAL VARIABLES

EXIT:
  MV  SP, BP           ; SET STACK POINTER TO PREVIOUS FRAME (FRAME POINTER)
  POP BP               ; POP FRAME POINTER FROM STACK
  POP AS               ; POP RETURN ADDRESS FROM STACK
  JR  AS               ; RETURN FROM SUBROUTINE

LV  RX, #Y:            ; LOAD LOCAL VARIABLE Y INTO RX
  LW  RX, BP, #Y

SV  RX, #Y:            ; STORE RX TO LOCAL VARIABLE Y
  SW  RX, BP, #Y
```

ARITHMETIC OPERATIONS

24-bit ADD (5 ops, 7 ops if carry out):

```
ADD24 AX, BX, CX, DX:

ADD BX, DX          ; ADD LOWER WORD
MV  AS, BX          ; COMPUTE CARRY
LR  AS, DX          ; AS = CARRY IN
ADD AX, AS          ; ADD CARRY IN
ADD AX, CX          ; ADD UPPER WORD
MV  AS, AX          ; COMPUTE CARRY  (optional)
LR  AS, CX          ; AS = CARRY OUT (optional)
```