

# porta\_glue\_coleco.v

---

## AUTHORS

---

JAY CONVERTINO

---

## DATES

---

2024/11/06

---

## INFORMATION

---

### Brief

---

Colecovision SGM glue logic chip

### License MIT

---

Copyright 2024 Jay Convertino

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

## CONSTANTS

---

### DEF\_RESET\_DELAY\_BIT

---

Number of bits for reset delay register

### DEF\_FB\_MONOSTABLE\_COUNT

---

delay till state is at 1 instead of 0 (its stable state) for feedback stable circuit

### DEF\_IRQ\_MONOSTABLE\_COUNT

---

delay till state is at 1 instead of 0 (its stable state) for the controller (spinner) generated interrupt.

## porta\_glue\_coleco

---

```
module porta_glue_coleco (  
  input  
  clk,  
  
  15:0]  
  A,  
  input  
  C1P1,  
  input  
  C1P2,  
  input  
  C1P3,  
  input  
  C1P4,  
  input  
  C1P6,  
  input  
  C1P7,  
  input  
  C1P9,  
  input  
  C2P1,  
  input  
  C2P2,  
  input  
  C2P3,  
  input  
  C2P4,  
  input  
  C2P6,  
  input  
  C2P7,  
  input  
  C2P9,  
  input  
  MREQn,  
  input  
  IORQn,  
  input  
  RFSHn,  
  input  
  M1n,  
  input  
  WRn,  
  input  
  RESETn_SW,  
  input  
  RDn,  
  
  7:0]  
  D,  
  output  
  CP5_ARM,  
  output  
  CP8_FIRE,  
  output  
  CS_h8000n,  
  output  
  CS_hA000n,  
  output
```

input [

inout [

```

    CS_hC000n,
    output
    CS_hE000n,
    output
    SND_ENABLEn,
    output
    ROM_ENABLEn,
    output
    RAM_CSn,
    output
    RAM_OEn,
    output
    CSWn,
    output
    CSRn,
    output
    WAITn,
    output
    RESETn,
    output
    RAM_MIRRORn,
    output
    INTn,
    output
    AS,
    output
    AY_SND_ENABLEn
)

```

Colecovision Super Game Module Glue Logic

## Ports

<b>clk</b> input	Clock for all devices in the core
<b>A</b> input[ 15: 0]	Address input bus from Z80
<b>C1P1</b> input	DB9 Controller 1 Pin 1
<b>C1P2</b> input	DB9 Controller 1 Pin 2
<b>C1P3</b> input	DB9 Controller 1 Pin 3
<b>C1P4</b> input	DB9 Controller 1 Pin 4
<b>C1P6</b> input	DB9 Controller 1 Pin 6
<b>C1P7</b> input	DB9 Controller 1 Pin 7
<b>C1P9</b> input	DB9 Controller 1 Pin 9
<b>C2P1</b> input	DB9 Controller 2 Pin 1
<b>C2P2</b> input	DB9 Controller 2 Pin 2
<b>C2P3</b> input	DB9 Controller 2 Pin 3
<b>C2P4</b> input	DB9 Controller 2 Pin 4
<b>C2P6</b>	DB9 Controller 2 Pin 6

input	
<b>C2P7</b> input	DB9 Controller 2 Pin 7
<b>C2P9</b> input	DB9 Controller 2 Pin 9
<b>MREQn</b> input	Z80 memory request input, active low
<b>IORQn</b> input	Z80 IO request input, active low
<b>RFSHn</b> input	Z80 Refresh input, active low
<b>M1n</b> input	Z80 M1 state, active low
<b>WRn</b> input	Z80 Write to bus, active low
<b>RESETn_SW</b> input	Input for reset switch
<b>RDn</b> input	Z80 Read from bus, active low
<b>D</b> inout[ 7: 0]	Z80 8 bit data bus, tristate IN/OUT
<b>CP5_ARM</b> output	DB9 Controller 1&2 ARM Select
<b>CP8_FIRE</b> output	DB9 Controller 1&2 FIRE Select
<b>CS_h8000n</b> output	Select when Z80 requests memory at h8000 (GAME CART), active low
<b>CS_hA000n</b> output	Select when Z80 requests memory at hA000 (GAME CART), active low
<b>CS_hC000n</b> output	Select when Z80 requests memory at hC000 (GAME CART), active low
<b>CS_hE000n</b> output	Select when Z80 requests memory at hE000 (GAME CART), active low
<b>SND_ENABLEn</b> output	SN76489 Sound chip enable, active low
<b>ROM_ENABLEn</b> output	Enable BIOS ROM, active low
<b>RAM_CSn</b> output	RAM chip select, active low
<b>RAM_OEn</b> output	RAM Ouput enable, active low
<b>CSWn</b> output	Chip Select Write for VDP, active low
<b>CSRn</b> output	Chip Select Read for VDP, active low
<b>WAITn</b> output	Wait state generator for Z80, active low
<b>RESETn</b> output	Timed reset generated by Logic, active low
<b>RAM_MIRRORn</b> output	Extended RAM, high is extended RAM, active low is mirrored.
<b>INTn</b> output	Interrupt generator for Z80, active low

**AS**                                      AY sound chip address(0)/data(1) select  
 output  
**AY\_SND\_ENABLEn**              AY sound enable, active low  
 output

## REGISTER INFORMATION

Core has 3 registers at the addresses that follow.

**SOUND\_CACHE**                      h51  
**RAM\_24K\_ENABLE**                  h53  
**SWAP\_BIOS\_TO\_RAM**              h7F

### SOUND\_CACHE

```
localparam SOUND_CACHE = 8'h51
```

Defines the address of r\_snd\_cache

SOUND CACHE REGISTER	
7:0	
CACHE LAST WRITE TO AY SOUND CHIP	

Cache Sound Chip as the SGM games read from it (Yamaha chip does not have a read like a GI does).

### RAM\_24K\_ENABLE

```
localparam RAM_24K_ENABLE = 8'h53
```

Defines the address of r\_24k\_ena

24K RAM ENABLE REGISTER	
7:1	0
ZERO	ENABLE 24K RAM, ACTIVE HIGH

Super Game Module 24K RAM enable using bit 0 (Active High)

### SWAP\_BIOS\_TO\_RAM

```
localparam SWAP_BIOS_TO_RAM = 8'h7F
```

Defines the address of r\_swap\_ena

SWAP BIOS TO RAM REGISTER			
7:4	3:2	1	0
ZERO	ONE	BIO TO RAM SWAP, ACTIVE LOW	ONE

Super Game Module BIOS to RAM swap on bit 1 (Active Low)

## r\_24k\_ena

```
reg [ 7:0] r_24k_ena = 0
```

register for RAM\_24K\_ENABLE See Also: [RAM\\_24K\\_ENABLE](#)

## r\_swap\_ena

```
reg [ 7:0] r_swap_ena = 8'h0F
```

register for 8K RAM/ROM swap See Also: [SWAP\\_BIOS\\_TO\\_RAM](#)

## r\_snd\_cache

```
reg [ 7:0] r_snd_cache = 0
```

register for SOUND\_CACHE See Also: [SOUND\\_CACHE](#)

## r\_int\_p1

```
reg r_int_p1 = 1'b0
```

Interrupt from player one control

## r\_int\_p2

```
reg r_int_p2 = 1'b0
```

Interrupt from player two control

## r\_wait

```
reg r_wait = 1'b0
```

Wait state generated register

## r\_reset\_counter

```
reg [ 9:0] r_reset_counter = 0
```

---

Timed reset counter

---

## r\_resetn

```
reg r_resetn = 0
```

Registered reset output, active low

---

## r\_mono\_count\_p1

```
reg [11:0] r_mono_count_p1 = 0
```

monostable circuit counters, player 1 AND

---

## r\_mono\_count\_p2

```
reg [11:0] r_mono_count_p2 = 0
```

monostable circuit counters, player 2 AND

---

## r\_mono\_count\_int\_p1

```
reg [ 5:0] r_mono_count_int_p1 = 0
```

monostable circuit counters, player 1 interrupt

---

## r\_mono\_count\_int\_p2

```
reg [ 5:0] r_mono_count_int_p2 = 0
```

monostable circuit counters, player 2 interrupt

---

## r\_mono\_p1

```
reg r_mono_p1 = 1'b0
```

Feedback from IRQ to controller 1 register

---

## r\_mono\_p2

```
reg r_mono_p2 = 1'b0
```

Feedback from IRQ to controller 2 register

---

## r\_ctrl\_fire

---

```
reg r_ctrl_fire = 1'b1
```

NAND Feedback Flip Flop FIRE select.

## r\_ctrl\_arm

---

```
reg r_ctrl_arm = 1'b0
```

NAND Feedback Flip Flop ARM select.

## ASSIGNMENT INFORMATION

---

How signals are created

## s\_ram\_csn

---

```
assign s_ram_csn = (
    s_y0_seln |
    r_swap_ena[1]
) & (s_ram2_csn | ~r_24k_ena[0]) & (s_ram1_csn | ~r_24k_ena[0]) & s_ram0_csn
```

RAM Chip select when address is requested (active low).

**(s\_y0\_seln | r\_swap\_ena[1])** address range starting at h0000, swap bios/rom bit is enabled (1 is disabled).

**(s\_ram1\_csn | ~r\_24k\_ena[0])** address range starting at h4000, 24k enable bit from register.

**(s\_ram2\_csn | ~r\_24k\_ena[0])** address range starting at h2000, 24k enable bit from register.

**s\_ram0\_csn** address range starting h6000, this is always an available range.

## RAM\_OEn

---

```
assign RAM_OEn = RDn | s_ram_csn
```

RAM Output enable when read is requested (active low).

**RDn** Z80 read request, active low.

**s\_ram\_csn** See Also: [s\\_ram\\_csn](#)

## RAM\_CSn

---

```
assign RAM_CSn = s_ram_csn
```

RAM Chip Select output assignment.

**s\_ram\_csn** See Also: [s\\_ram\\_csn](#)

## RAM\_MIRRORn

---



```
assign RAM_MIRRORn = (
  r_24k_ena[0] |
  r_swap_ena[1]
)
```

RAM Mirror enable. Output to AND gates that block address lines (active low)

**r\_24k\_ena[0]** If 24k ram extension is disabled, enable ram mirror  
**r\_swap\_ena[1]** If ram/bios swap is disabled, enable ram mirror.

## ROM\_ENABLEn

```
assign ROM_ENABLEn = (
  s_y0_seln |
  r_swap_ena[1]
)
```

ROM enable (active low).

**s\_y0\_seln** Only select ROM when address range h0000 is enabled.  
**r\_swap\_ena[1]** If ram/bios swap is disabled, enable ROM.

## DECODER INFORMATION FOR U5

How address decoder is created.

### s\_enable\_u5

```
assign s_enable_u5 = (
  RFSHn &
  MREQn
)
```

Enable the the decoder, duplicates U5 functionality from colecovision. always 1, RFSH is a double inversion on coleco (inverter + 138 internal)

**RFSHn** Z80 Refresh line, when not in refresh enable is active.  
**MREQn** When the MREQn is active then encoder is enabled.

### s\_y0\_seln

```
assign s_y0_seln = ~(
  A[14] &
  A[13]
)
s_enable_u5 & ~A[15] & ~
```

Address h0000, ROM/RAM

**s\_enable\_u5** Enable decoder

**A[15:13]**      Address lines used for select lines.

## s\_ram2\_csn

---

```
assign s_ram2_csn = ~(  
    A[14] &  
    A[13]  
    )  
s_enable_u5 & ~A[15] & ~
```

Address h2000, RAM

**s\_enable\_u5**      Enable decoder

**A[15:13]**      Address lines used for select lines.

## s\_ram1\_csn

---

```
assign s_ram1_csn = ~(  
    A[14] &  
    A[13]  
    )  
s_enable_u5 & ~A[15] &  
~
```

Address h4000, RAM

**s\_enable\_u5**      Enable decoder

**A[15:13]**      Address lines used for select lines.

## s\_ram0\_csn

---

```
assign s_ram0_csn = ~(  
    A[14] &  
    A[13]  
    )  
s_enable_u5 & ~A[15] &
```

Address h6000, RAM

**s\_enable\_u5**      Enable decoder

**A[15:13]**      Address lines used for select lines.

## CS\_h8000n

---

```
assign CS_h8000n = ~(  
    A[14] &  
    A[13]  
    )  
s_enable_u5 & A[15] & ~
```

Address h8000, Game ROM bank select.

**s\_enable\_u5**      Enable decoder

**A[15:13]** Address lines used for select lines.

## CS\_hA000n

---

```
assign CS_hA000n = ~(  
    A[14] &  
    A[13]  
)  
s_enable_u5 & A[15] & ~
```

Address hA000, Game ROM bank select.

**s\_enable\_u5** Enable decoder

**A[15:13]** Address lines used for select lines.

## CS\_hC000n

---

```
assign CS_hC000n = ~(  
    A[14] &  
    A[13]  
)  
s_enable_u5 & A[15] & ~
```

Address hC000, Game ROM bank select.

**s\_enable\_u5** Enable decoder

**A[15:13]** Address lines used for select lines.

## CS\_hE000n

---

```
assign CS_hE000n = ~(  
    A[14] &  
    A[13]  
)  
s_enable_u5 & A[15] &
```

Address hE000, Game ROM bank select.

**s\_enable\_u5** Enable decoder

**A[15:13]** Address lines used for select lines.

## DECODER INFORMATION FOR U6

---

How address decoder is created

## s\_enable\_u5

---

Enable the the decoder, duplicates U6 functionality from colecovision.

**A[7]** Address IO range h80 to hFF

**IORQn** When the IORQn is active then encoder is enabled.

## s\_ctrl\_en\_2n

---

```
assign s_ctrl_en_2n = ~(  
    A[5] &                                     s_enable_u6 & ~A[6] & ~  
    WRn  
);
```

h80 PORT IO for controller Fire Select

**s\_enable\_u6**     Enable decoder

**A[6:5]**           Address lines used for select lines.

**WRn**             Select write or read.

## CSWn

---

```
assign CSWn = ~(  
    A[5] &                                     s_enable_u6 & ~A[6] & ~  
    WRn  
);
```

hBE PORT IO for VDP write

**s\_enable\_u6**     Enable decoder

**A[6:5]**           Address lines used for select lines.

**WRn**             Select write or read.

## CSRn

---

```
assign CSRn = ~(  
    A[5] &                                     s_enable_u6 & ~A[6] & ~  
    WRn  
);
```

hBF PORT IO for VDP read

**s\_enable\_u6**     Enable decoder

**A[6:5]**           Address lines used for select lines.

**WRn**             Select write or read.

## s\_ctrl\_en\_1n

---

```
assign s_ctrl_en_1n = ~(  
    A[5] &                                     s_enable_u6 & A[6] & ~  
    WRn  
);
```

hC0 PORT IO for controller ARM select

**s\_enable\_u6**    Enable decoder  
**A[6:5]**        Address lines used for select lines.  
**WRn**            Select write or read.

## SND\_ENABLEn

```
assign SND_ENABLEn = ~(
    A[5] &
    WRn
)
```

s\_enable\_u6 & A[6] &

hFF PORT IO for sound enable.

**s\_enable\_u6**    Enable decoder  
**A[6:5]**        Address lines used for select lines.  
**WRn**            Select write or read.

## s\_ctrl\_readn

```
assign s_ctrl_readn = ~(
    A[5] &
    WRn
)
```

s\_enable\_u6 & A[6] &

hFC/FF PORT IO for controller read

**s\_enable\_u6**    Enable decoder  
**A[6:5]**        Address lines used for select lines.  
**WRn**            Select write or read.

## DECODER INFORMATION FOR SUPER GAME MODULE

How address decoder is created for Super Game Module

**SGM IO REG**    Clocked IO decoder for Super Game Module.

## AS

```
assign AS = (
    A[7:0]
    =
    = 8'h50 & ~IORQn & ~WRn ? 1'b0 : 1'b1
)
```

h50 is the address select, when selected its in data mode

**A[7:0]**        If address matches h50, enable  
**IORQn**        Active IO request, enable  
**WRn**          Z80 write is active, enable

## AY\_SND\_ENABLEn

---

```
assign AY_SND_ENABLEn = (  
  A[7:1]  
  =  
  = 7'b0101000 & ~IORQn & ~WRn ? 1'b0 : 1'b1  
  )
```

match both h50 and h51 by ignoring bit 0. Enable AY sound chip.

**A[7:0]**     If address matches h50 or h51, enable

**IORQn**     Active IO request, enable

**WRn**        Z80 write is active, enable

## AY\_SND\_ENABLEn

---

read cached register from previous write (AY emulation).

**A[7:0]**     If address matches h52, enable

**IORQn**     Active IO request, enable

**RDn**        Z80 read is active, enable

## CONTROLLER REGISTER READ

---

How to read controller inputs for player 1 and 2, works with roller and standard gamepads.

## CP5\_ARM

---

```
assign CP5_ARM = r_ctrl_arm
```

Activate ARM portion of controllers.

**r\_ctrl\_arm**     See Also: [r\\_ctrl\\_arm](#)

## CP8\_FIRE

---

```
assign CP8_FIRE = r_ctrl_fire
```

Activate FIRE portion of controllers.

**r\_ctrl\_fire**     See Also: [r\\_ctrl\\_fire](#)

## D[0]

---

```
assign D[0] = (  
  C1P1  
  :  
  1'bz  
  )  
  ~s_ctrl_readn & ~A[1] ?
```

Data bit zero for P1

**s\_ctrl\_readn**    See Also: **s\_ctrl\_readn**, read when active low  
**A[1]**            Address bit 1 is 0, read

## D[1]

---

```
assign D[1] = (
C1P4
:
1'bz
)
-s_ctrl_readn & ~A[1] ?
```

Data bit one for P1

**s\_ctrl\_readn**    See Also: **s\_ctrl\_readn**, read when active low  
**A[1]**            Address bit 1 is 0, read

## D[2]

---

```
assign D[2] = (
C1P2
:
1'bz
)
-s_ctrl_readn & ~A[1] ?
```

Data bit two for P1

**s\_ctrl\_readn**    See Also: **s\_ctrl\_readn**, read when active low  
**A[1]**            Address bit 1 is 0, read

## D[3]

---

```
assign D[3] = (
C1P3
:
1'bz
)
-s_ctrl_readn & ~A[1] ?
```

Data bit three for P1

**s\_ctrl\_readn**    See Also: **s\_ctrl\_readn**, read when active low  
**A[1]**            Address bit 1 is 0, read

## D[4]

---

```
assign D[4] = (
r_mono_p1
:
1'bz
)
-s_ctrl_readn & ~A[1] ?
```

Data bit one for P1

**s\_ctrl\_readn**      See Also: **s\_ctrl\_readn**, read when active low

**A[1]**                Address bit 1 is 0, read

## D[5]

---

```
assign D[5] = (
    C1P7
    :
    1'bz
)
```

~s\_ctrl\_readn & ~A[1] ?

Data bit five for P1

**s\_ctrl\_readn**      See Also: **s\_ctrl\_readn**, read when active low

**A[1]**                Address bit 1 is 0, read

## D[6]

---

```
assign D[6] = (
    C1P6
    :
    1'bz
)
```

~s\_ctrl\_readn & ~A[1] ?

Data bit six for P1

**s\_ctrl\_readn**      See Also: **s\_ctrl\_readn**, read when active low

**A[1]**                Address bit 1 is 0, read

## D[7]

---

```
assign D[7] = (
    s_int_p1
    :
    1'bz
)
```

~s\_ctrl\_readn & ~A[1] ?

Data bit seven for P1

**s\_ctrl\_readn**      See Also: **s\_ctrl\_readn**, read when active low

**A[1]**                Address bit 1 is 0, read

## s\_int\_p1

---

```
assign s_int_p1 = ~(
    r_mono_p1 &
    C1P9
)
```



generate interrupt for player one

**r\_mono\_p1** See Also: **r\_mono\_p1**, RC TL emulation

**C1P9** Input from controller port. Roller controller only.

## D[0]

---

```
assign D[0] = (
    C2P1                                     ~s_ctrl_readn & A[1] ?
    :
    1'bz
)
```

Data bit zero for P2

**s\_ctrl\_readn** See Also: **s\_ctrl\_readn**, read when active low

**A[1]** Address bit 1 is 1, read

## D[1]

---

```
assign D[1] = (
    C2P4                                     ~s_ctrl_readn & A[1] ?
    :
    1'bz
)
```

Data bit one for P2

**s\_ctrl\_readn** See Also: **s\_ctrl\_readn**, read when active low

**A[1]** Address bit 1 is 1, read

## D[2]

---

```
assign D[2] = (
    C2P2                                     ~s_ctrl_readn & A[1] ?
    :
    1'bz
)
```

Data bit two for P2

**s\_ctrl\_readn** See Also: **s\_ctrl\_readn**, read when active low

**A[1]** Address bit 1 is 1, read

## D[3]

---

```
assign D[3] = (
    C2P3                                     ~s_ctrl_readn & A[1] ?
    :
    1'bz
)
```

---

Data bit three for P2

**s\_ctrl\_readn**      See Also: **s\_ctrl\_readn**, read when active low

**A[1]**                Address bit 1 is 1, read

## D[4]

---

```
assign D[4] = (
    r_mono_p2
    :
    1'bz
)
```

~s\_ctrl\_readn & A[1] ?

Data bit four for P2

**s\_ctrl\_readn**      See Also: **s\_ctrl\_readn**, read when active low

**A[1]**                Address bit 1 is 1, read

## D[5]

---

```
assign D[5] = (
    C2P7
    :
    1'bz
)
```

~s\_ctrl\_readn & A[1] ?

Data bit five for P2

**s\_ctrl\_readn**      See Also: **s\_ctrl\_readn**, read when active low

**A[1]**                Address bit 1 is 1, read

## D[6]

---

```
assign D[6] = (
    C2P6
    :
    1'bz
)
```

~s\_ctrl\_readn & A[1] ?

Data bit six for P2

**s\_ctrl\_readn**      See Also: **s\_ctrl\_readn**, read when active low

**A[1]**                Address bit 1 is 1, read

## D[7]

---

```
assign D[7] = (
    s_int_p2
    :
```

~s\_ctrl\_readn & A[1] ?

```
1'bz  
)
```

Data bit seven for P2

**s\_ctrl\_readn**     See Also: **s\_ctrl\_readn**, read when active low

**A[1]**                Address bit 1 is 1, read

## s\_int\_p2

```
assign s_int_p2 = ~(  
  r_mono_p2 &  
  C2P9  
)
```

generate interrupt for player one

**r\_mono\_p1**     See Also: **r\_mono\_p1**, RC TL emulation

**C2P9**            Input from controller port. Roller controller only.

## INTn

```
assign INTn = ~(  
  r_int_p1 |  
  r_int_p2  
)
```

INTn is generated by monostable circuit based on NAND outputs.

**r\_int\_p1**     See Also: **r\_int\_p1**, RC TL emulation

**r\_int\_p2**     See Also: **r\_int\_p2**, RC TL emulation

## CIRCUIT EMULATION

Everything below emulates a part of the circuit that uses some sort of linear/non-linear components to perform its task. Things such as RC reset circuits, RC interrupts, IRQ and others. See this source file for details.

**WAIT GENERATE**     Generate wait states for the Z80 proccessor

**RESET GENERATE**    Generate a timed reset for the CPU/VDP/ETC.

**TL RC RESET**        Generate a interrupt for a monostable circuit that will trigger a 1 for a short duration.

**CONTROLLER NAND**    Controller NAND Latch FIRE/ARM emulation.

**NAND IRQ PULSE**     Controller bit 4 is a pulse that represents the spinner state.