CPUCORE 2.00 (TM)

gferrante@opencores.org

Overview

Usage: **cpucore** -c CONFIG_FILE -d HW_TARGET_DIRECTORY

```
      (ex.) cpucore -c cpu4bit.dat -d cpu4bit
```

Input files: configuration file, target hardware directory

Output files:

```
      - cpu#_utils.vhd:       CPU vhdl utility package (first)
      - cpu#_iu.vhd:          CPU vhdl Instruction Unit
      - cpu#_cu.vhd:          CPU vhdl Control Unit
      - cpu#_du.vhd:          CPU vhdl Data Unit
      - cpu#_oa.vhd:          CPU vhdl Over Addressing Unit
      - cpu#_wd.vhd:          CPU vhdl Write Delay Unit
      - cpu#.vhd:             CPU vhdl Structure (top)
```

Defines

1) Instruction BUS:

```
      IWL   = Instruction Word Length
      IAL   = Instruction Address Length
      IRL   = Instruction ROM Length
```

2) Data BUS

```
      DBL   = Data Bus Length
      DBAL  = Data Bus Address Length
      DBRL  = Data Bus RAM Length
```

3) Stack

```
      ST    = Stack Type
      HSD   = Hardware Stack Deep
```

4) Options

```
      I     = Interrupt
      IA    = Indirect Address
      IVL   = Instruction Variable Latency
      DVL   = Data Variable Latency
```

5) Auxiliary

```
      CBUS  = Code BUS         = 8
      MBUS  = Minimum BUS      = 4
      IBUS  = Internal BUS     = IWL - MBUS
```

Rules

```
1) (IWL  >= CBUS)
2) (DBL  >= MBUS) and (DBL <= (2 * IBUS))
3) (IAL  >= IBUS) and (IAL <= (DBL + IBUS))
4) (DBAL >= IBUS) and (DBAL <= (DBL + IBUS))
5) (DBRL <= DBAL)
6) (IRL  <= IAL)
```

Architecture

Cpucore (TM) is a RISC 3-stage pipeline CPU with separate data and instruction bus generator.
The cpu generally executes every instruction using one clock cycle.
Branches (when skip occurs) and RAW (Read After Write) hazards, with 2 memory access, may need one more execution cycle.

Cpuasm (TM) instructions are divided into control classes: every instruction within a class has similar behavior and pipeline datapath.
Control classes type are:

1) **TC_OTH**: logical/conditional_branch class instructions (ex. nop, not, ror, rol, shl, shr, cla, skz, sknz, skc, sknc): accumulator/s is/are used.
2) **TC_LDA**: memory class instructions (ex. lda, add, and, sub, or, xor): accumulator/s and data_in bus are used.
3) **TC_DAT**: immediate class instructions (ex. ldai, addi, andi, subi, ori, xori): accumulator/s and instruction argument are used.
4) **TC_STA**: storage class instructions (ex. sta): accumulator/s and data_out bus are used.
5) **TC_RTI**: return from interrupt class instructions (ex. rti)
6) **TC_RTS**: return from subroutine class instructions (ex. rts)
7) **TC_JMP**: jump class instructions (ex. jmp)
8) **TC_JMS**: jump to subroutine class instructions (ex. jms)

Within the same control class every instruction is classified with a data class(). **Cpucore** permits easy extension/redefinition of its instruction set (custom instructions), but the instruction classification into control classes must be preserved.

VHDL CORE PROCESSOR

Cpucore (TM) will generate in the TARGET directory the following files:

1) **CPU_UTILS**: Cpu Define Constants

2) **CPU_IU**: Cpu Instruction Unit

- generate instruction address for internal memory code
- increment program counter during normal operations
- save/load program counter into/from internal/external stack (interrupt or subroutine jump/return)
- stall addressing (in case of conflicts on resources access (RAW) and variable latency access)

3) **CPU_CU**: Cpu Control Unit

- decode instructions (detect control and data classes)
- manage variable latency access
- manage interrupt request

- generate memory/register write and read enable
- invalid current instruction (RAW condition, skip ...)
- select next instruction addressing (pc_mux)

4) **CPU_DU**: Cpu Data Unit

- arithmetic and logical operations (ALU)
- test branch conditions (skip)

5) **CPU_OA**: Cpu Over Addressing Unit

It hooks and modify some external data memory access.
This permits for base core CPU features and bus extension:

- **Indirect addressing**:

  Operations on 3 particular memory address [0], [1] and [2] permits realization of data memory indirect addressing. Any external memory operation at addresses [1] and [2] is replaced with the same operation on internal registers (IND_REG) and (IND_INC) respectively.
  Every operation on external memory at address [0] (IND_ADD) is replaced with the same operation on external memory at the address contained into the IND_REG register and its (IND_REG) value will also be incremented by the content of the IND_INC register.
  This is implemented when (IA = 1)

- **Instruction Memory Extension**:

  An external memory operation at address [3] is replaced with the same operation on internal ISTR_PAGE register. This register allows to extend instruction memory access; it preserves the most significant part of instruction address and becomes active in the next jump, two clock periods after setting the value.
  This is implemented when (IAL > IBUS)

- **Data Memory Extension**:

  An external memory operation at address [4] is replaced with the same operation on internal DATA_PAGE register. This register allows to extend data memory access; it preserves the most significant part of the data address.
  This is implemented when (DBAL > IBUS)

- **Data Bus Expansion**:

  An external memory operation at address [5] is replaced with the same operation on internal DATA_EXP register. This register allows to extend data bus size; it keeps the most significant data of an immediate data operation.
  This is implemented when (DBL > IBUS)

6) **CPU_WD**: Cpu Write Delay Unit

- delays data address and write enable signals for one clock cycle

7) **CPU**: Microcontroller Core Unit

CPU core: top level hierarchical structural model

INTERRUPTS

Cpucore (TM) permits to manage a one-level interrupt input.
As soon as the "int_in" line becomes active (1):

- the instruction unit saves the current program counter to the internal stack (ST = 1 and HSD > 0) or just pushes it onto an external stack (ST = 0)
- internal registers (accumulator/s, instruction addr. extension, data addr. extension and data expansion) are saved
- a jump to the "all one minus '8'" address (to update interrupt addr. page) is performed
- moves to the "interrupt" state

Only after a "rti" (return from interrupt) class instruction, the system will come back to the normal state by restoring the internal registers, popping addresses from stack and returning to check against the "int_in" level line.
This is implemented when (I = 1)

INSTRUCTION VARIABLE LATENCY

Cpucore (TM) has one level instruction wait line (iwait_in).
The processor continues to perform the same memory access instruction until "iwait_in" line is active (1). This allows to delay memory access instructions (instruction cache miss, Harvard to Von-Neuman architecture). This is implemented when (IVL = 1)

DATA VARIABLE LATENCY

Cpucore (TM) has one level data wait line (dwait_in).
The processor continues to perform the same data memory access until the "dwait_in" line is active (1). This allows to delay data memory accesses (data cache miss, wait state generator). This is implemented when (DVL = 1)

Memory/Register interface

The Instruction/data memory interface needs to use internal/external synchronous, single clock, memories (not registered q output ports).

Input/Output registers are mapped to the assembler __reg and __ram zone.

1) Write to register

Write data_out on rising clk_in edge when:

- daddr_out matches register address
- ndwe_out = '0'

2) Read from register

Read data_in on rising clk_in edge when:

- daddr_out matches register address
- ndre_out = '0'

It's necessary to implement an input multiplexer in order to use more memories/registers input (see Tutorial for interfacing examples).

SIGNALS DIAGRAM

Instruction signals:

Clk_in	RE	RE	RE	RE	RE	RE	RE	RE
Nreset_in	1	1	1	1	1	1	1	1
Iwait_in	0	0	1	0	1	1	0	0
Idata_in	VI|i	VI|i+1	II	VI|i+2	II	II	VI|i+3	VI|i+4
Iaddr_out	AI|i+1	A|i+2	A|i+2	A|i+3	A|i+3	A|i+3	A|i+4	A|i+5

RE = Rise Edge
VI|i = Valid instruction (i)
II = Invalid instruction
AI|i = Address instruction (i)

Read data signals:

Clk_in	RE	RE	RE	RE	RE	RE	RE	RE
Nreset_in	1	1	1	1	1	1	1	1
Dwait_in	0	0	0	0	1	1	0	0
Data_in	-	-	VD|i	-	ID	ID	VD|j	-
Data_out	-	-	-	-	-	-	-	-
Adaddr_out	-	AD|i	-	AD|j	-	-	-	-
Daddr_out	-	AD|i	-	AD|j	-	-	-	-
Nadwe_out	1	1	1	1	1	1	-	-
Ndwe_out	-	1	1	1	1	1	1	-
Ndre_out	-	0	-	0	1	1	-	-

RE = Rise Edge
VD|i = Valid data (i)
ID = Invalid data
AD|i = Address data (i)
- = Don't care

Write data signals:

Clk_in	RE	RE	RE	RE	RE	RE	RE	RE
Nreset_in	1	1	1	1	1	1	1	1
Dwait_in	0	0	0	0	1	1	0	0
Data_in	-	-	-	-	-	-	-	-
Data_out	-	VD|i	-	-	VD|j	VD|j	VD|j	-
Adaddr_out	AD|i	-	-	AD|j	-	-	-	-
Daddr_out	-	AD|i	-	-	AD|j	-	-	-
Nadwe_out	0	-	-	0	1	1	-	-
Ndwe_out	-	0	-	-	0	1	1	-
Ndre_out	1	1	-	1	1	1	1	-

RE = Rise Edge
VD|i = Valid data (i)
ID = Invalid data
AD|i = Address data (i)
- = Don't care

How to set stack depth

1) No interrupt support (I = 0)

    Stack depth (HSD) >= (max. number of subroutine nested calls within the main code)

2) Interrupt support (I = 1)

    Stack depth (HSD) >= (max. number of subroutine nested calls within the main code) + (max. number of subroutine nested calls within interrupt code) + 1

Custom Instructions

Cpucore (TM) generates a custom cpu that can be used as a template for instruction customizations. At the moment, this process requires direct changing of the VHDL generated code, therefore every new HWBuild execution will overwrite these changes. To avoid loosing every modification made to the code, I suggest to use a different cpu ID for custom and template cpu outputs. The cpu architecture and files structure allows to insert a lot of customizations just by changing 2 process in 3 VHDL files.

Customization flow:

1) From the program code to execute, decide which instructions to add/change
2) From the total number of instructions, compute the number of bits required to encode them (ENCODE_BIT)
3) Compute the maximum number of internal registers required (INT_REG), the maximum data bus input and output size (IN_BUS, OUT_BUS)
4) For each instruction define its the class and encoding
5) Change these parameters in the custom file cpu#_utils.vhd:

    - CODE_LEN = ENCODE_BIT [>= 4]
    - ICODE_LEN = ENCODE_BIT [>= 4]
    - DCLASS_LEN = ENCODE_BIT [>= 4]
    - MBUS_LEN = min(ISTR_LEN - ENCODE_BIT, DATA_LEN)
    - ACC_NUM = INT_REG [>= 1]
    - DIN_LEN = IN_BUS [>= DATA_LEN]
    - DOUT_LEN = OUT_BUS [>= DATA_LEN]
    - Update Cpu_code, Cpu_icode, Cpu_dclass   CONSTANTS
    - Update debug TYPE

6) Update the custom file cpu#_cu.vhd "CU_DECODE: process" to decode the new (modified) instructions
7) Update the custom file cpu#_du.vhd "DU_ALU: process" to execute the new (modified) instructions
8) Update the template file cpu#.ist to generate the correct assembler instruction set.

Example:

Let’s define a CPU with the following custom instructions:

1) Sum with sign [ADDS]
2) Skip if overflow [SKV]
3) Move acc(1) to acc(0) [MOVHL]

1)**ADDS**: sum with sign: acc(0) = acc(0) + CONSTANT/VARIABLE; acc(1) <= status register

```
       acc(0)     <=    (('0' & signed(acc_c(0)((DATA_LEN - 1) downto 0))) +
('0' & signed(data_x)));

      if ((acc_c(0)(DATA_LEN - 1) /= data_x(DATA_LEN - 1)) or
((acc_c(0)(DATA_LEN - 1) = acc(0)(DATA_LEN - 1)))) then
            acc(1)(DATA_LEN) <= '0';      -- NO OVERFLOW
      else
            acc(1)(DATA_LEN) <= '1';      -- OVERFLOW
      end if;
      if (class_cu  = TC_DAT) then
            code  <=    ADDSI_I;    -- CONSTANT OPERAND
      else
            code  <=    ADDS_I;     -- VARIABLE OPERAND
      end if;
```

2)**SKV**: skip one instruction if overflow

```
      if (acc_c(1)(DATA_LEN) = '1') then skip_l <= '1';
      else  skip_l <= '0';
      end if;
      code  <=    SKV_I;
```

3)**MOVHL**: move acc(1) to acc(0)

```
      acc(0)      <=    acc(1);
      code        <=    MOVHL_I;
```

Notes:

- It is possible create instructions capable of executing multiple internal registers operations in one single clock cycle.
- Increasing the input data bus allows to process multi-operands instructions
- Increasing the output data bus size allows to insert multi-cycle/DSP instructions (ex. 1 more output bit can be used as dwait_in signal)
- Changing the instructions encoding size could change data expansion register dimension, data/instruction page number and size

License