**Debugging Codasip Projects with --info**

Debugging is a crucial component of successfully completing the Class Project, particularly for the CA model. The Codasip design includes a very handy debug facility which allows selective printing of important debug information. Although the Codasip environment provides the capabilities for monitoring signals and registers, debug printing is a much more efficient and easily used mechanism in many cases. Explicit print statements may also be used, but that process requires the hardware project be rebuilt after every modification. The internal debug print statements can be enabled simply at run time.

1. **The - -info Function**

Codasip maintains an internal variable called “info”, which is used to selectively enable a variety of debug print functions. The default info value is 0, which enables no printing. Info may be set to another value in the debug configuration:

1. Execute Run -> Debug Configurations.
2. Select the desired Project, Application and Debugger as defined in Phase 4.
3. Select the (x) = Arguments tab.
4. In the Simulator arguments area, enter “- -info N” (no quotes, no space between the dashes or between the dash and info) where N is a decimal number which is the desired value of info. Multiple values may be specified by separating them with commas (and no spaces), and all of the specified prints will occur. For example, - -info 6,7,11 enables all of the proint statements selected by info values of 6, 7 or 11.
5. Click Debug to run the simulation in the normal way.
6. The info value will remain as it is set whenever this debug configuration is run.

Figure 1 below shows an example which sets info to 6.

The info value is used by several functions, particularly codasip\_info and codasip print. The first parameter of these functions contains the info value which enables the print operation. The values for info are defined in debug.hcodal, and additional values can be added there as desired.

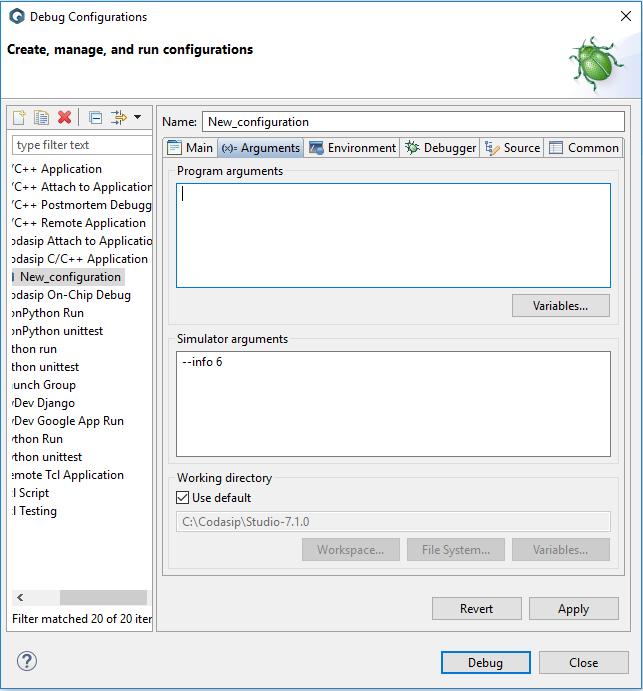


Figure 1

There are a number of predefined - -info values. The following sections will describe each value, present examples of the print statements which are enabled, and provide suggestions as to where the particular value may be useful in debugging.

1. **- -info Display Timing**

The cycle accurate (CA) model in Codasip executes in a sequence of clock cycles, clocking each of the registers (including the pipeline registers) of every pipeline stage at the end of each cycle. The - -info prints occur in the print\_pipeline function, which is executed at the end of each pipeline stage just before all of the registers are updated. This is a convenient place to observe signals, since all of the Codasip combinational code has been executed at this point and any register input values displayed will be transferred to the register output immediately after the print occurs.

1. **Pipeline Debug (- -info 6)**

The - -info 6 function displays the instruction in each of the four pipeline stages ID, EX, ME and WB. This is a very valuable display since it shows the complete pipeline, which makes it useful in gaining a clear understanding of the pipeline behavior. shows a typical display. The addi x10, x0, 1 instruction is in the WB stage, the addi x2, x0, 10 instruction is in the ME stage and so on. The stage is displayed first, and “(P)” indicates this is the pipeline stage. The number after the colon is the address of this instruction in hex, and the number after the last comma is the actual instruction in hex. This particular display shows the instructions at addresses 0, 4, 8 and c – the first four instructions in a test.

A picture containing keyboard

Description automatically generated

Figure 2

One potential issue with this display is that the offset displayed for a branch or jump (the 0x250 value in the first line of ) is neither the target address nor the branch offset value – this is due to a bug in Codasip.

If any pipeline stages are stalled or cleared by the assertion of s\_STG\_stall or s\_STG\_clear, - -info 6 will print a message for each stage as shown in . This example shows the pipeline flush on a taken branch or jump.

A picture containing orange, room, man, holding

Description automatically generated

Figure 3

When pipeline stages are cleared, the display is as shown in . In the ID stage a real nop (addi x0, x0, 0) is inserted. Other stages are cleared to all zeroes, as indicated by the .bit 8, 0 notation.

A screenshot of a cell phone

Description automatically generated

Figure 4

1. **Register File Debug (- -info 3)**

It is often important to understand exactly what is being written into the Register File, since many different instructions load a destination register. The - -info 3 function shows exactly what is written (or not) to the Register File in the current cycle. If a write occurs, the display shown in Figure 5 is printed. The “WB” indicates that the signals are related to the instruction in the WB stage, and the “(RF)” indicates that this is the Register File operation. In this particular case, the data value 0x32 will be written into register x10 on the next clock edge. If no write occurs, nothing will be printed.



Figure 5

1. **ALU Debug (- -info 7)**

The ALU is used in almost every instruction, so being able to see its inputs and outputs is extremely useful, particularly when Forwarding is occurring as in Phase 6. The - -info 7 function displays these values for the ALU as shown in . “EX” indicates that the signals are from the instruction in the EX stage, and “(ALU)” indicates that this is the ALU display. The ALU output is the signal s\_ex\_alu. The OP signal is r\_ex\_aluop, which indicates what operation the ALU is performing. In the example, OP = 10 indicates the ALU\_SLLI function is executed (ox6aa is 0x355 shifted left by 1). NOTE: the signal names of the ALU inputs may change from Phase to Phase based on the schematic implementation, so ca\_utils.codal must be updated whenever these signals change.



Figure 6

1. **Branch Debug (- -info 11)**

The branch decisions are made in the ME stage, so - -info 11 is very useful in debugging branches. When the instruction in the ME stage is a branch or jump, the information shown in is displayed. BROP is the value of r\_me\_brnchop, which indicates the type of branch decision being made. ZERO is the value of r\_me\_zero, which is the pipelined version of the zero detection in the previous ALU cycle. SEL is the value of s\_me\_pcsrc, which will be 1 if the branch is taken and 0 if the branch is not taken. ADDR is the branch address r\_me\_bradd. In this example, the branch will be taken and the target address is 0x578.



Figure 7

This display is important when debugging the implementation of branches and jumps in Phase 7.

1. **Memory Address Debug (- -info 8)**

The memory address is generated by the instruction in the EX stage, and - -info 8 is important in understanding the memory operation, and in particular the behavior of the MEMCTL block. When a load or store instruction is in the EX stage, the information shown in is displayed. ADDR is the value of s\_ex\_aluaddr, which is the address input to the memory address function. OPIN is the value of r\_ex\_memop, the memory operation input to MEMCTL. OPOUT is the value of s\_ex\_memop, the memory operation which is the operation input to the memory address function. RESP is the value of s\_ex\_resp, which generates the stall from MEMCTL. CNT is the value of r\_me\_memcnt, the current value of the counter in MEMCTL. The memory operation will actually occur when OPOUT is not 0 (i.e. it is not MEM\_NOP).



Figure 8

1. **Memory Data Debug (- -info 9)**

When a load or store instruction is in the ME stage the - -info 9 display shows information about the memory data as shown in . OP is the memory operation r\_me\_memop (which is the pipelined version of OPOUT from the previous cycle), RDATA is the read data s\_ex\_memdat, WDATA is the write data r\_me\_wtdat and RESP is s\_me\_resp, the ME stage stall signal. Note that the read data is only valid on a read operation, and the write data is only valid on a write operation.



Figure 9

1. **Using Multiple - -info Values**

below shows a display when several - -info values are enabled – in this case at least - -info 3,6,7,11. The (P) lines show the pipeline from - -info 6. The EX(ALU) line from - -info 7 shows the ALU inputs and outputs for the instruction in the EX stage, in this case the bne x8, x9, 0x7c8 instruction at address 0x584. The ME(BR) line from - -info 11 shows the branch signals for the instruction in the ME stage, in this case the bne x13, x5, 0xaf8 instruction at address 0x580. The WB(RF) line from - -info 3 shows the Register File write signals for the instruction in the WB stage, in this case the add x13, x13, x11 instruction at address 0x57c. Stepping through multiple cycles allows an easy understanding of how the pipeline flow works.

A screenshot of a cell phone

Description automatically generated

Figure 10

1. **Integrating Hardware Debugging with - -info**

In many cases the - -info displays will be sufficient to debug a problem, but in many other cases real hardware debug will be required. Even in these cases, - -info provides a lot of information which can quickly pinpoint where to debug in the hardware. In order to use these two methods together, it is critical to understand the timing relationship between them.

In hardware debug, we set a breakpoint at some point and then observe signals by hovering over them with the mouse. If the pipeline stage where the error is occurring is clear, the best point to set the breakpoint is on the STG\_output() call in that stage. At that point all combinational signals in that stage have been evaluated, so when observed they all hold their values in the same cycle. The advantage of this approach is that when we hit the breakpoint, that stage file is displayed so we can then directly observe signals. Note, however, that if we then switch to another stage the signals may or may not be in the same cycle. If we switch to a later pipeline stage (e.g. WB is later than ME), the signals are in the same cycle because we evaluate the stages in the reverse order. If we switch to an earlier stage (e.g. ID is earlier than EX), the signals will have values from the previous cycle since that stage has not been executed yet.

The - -info displays all show the signal values at the end of the cycle, since the prints happen in the print\_pipeline routine. This means that if a breakpoint is set in any stage, the displays will not yet have been printed. This means that if the pipeline display shows an instruction in the EX stage, a hardware breakpoint in the EX stage will show signal values from the following cycle. If it is desired to observe a particular instruction in the EX stage, the breakpoint must be hit when that instruction is in the ID stage of the pipeline display.

It is also possible to set the breakpoint at the end of the cycle, on the “if (s\_if\_stall)” statement in ca\_pipe\_control. At that point all stages have been evaluated, so that all combinational signals hold the value in the current cycle. This avoids confusion but requires more switching between files. The other advantage of setting the breakpoint here is that the - -info prints will have already occurred in print\_pipeline() and synchronizing the hardware signals to the - -info displays is trivial.