HANDEL-C DEVELOPMENT.

(LABORATORY II)

2/25/2004

Sergey Averchenkov

CS345, Queens College, N.Y.

ABSTRACT:

Studied more advanced concepts of Handel-C programming by developing an application that utilizes concepts such as parallelism.

INTRODUCTION:

The reason for this lab was to gain deeper understanding of Handel-C. In the processing of developing this Handel-C application we became more familiar with DK design suite and issues related to differences between software simulation and hardware implementation.

METHOD:

- **Created a new workspace**:
  - Started the DK IDE.
  - File → New. Created a new blank workspace with the name "Laboratory II," under "My Projects" directory. Saved the workspace and exited DK.
  - Used Explorer to verify that the Laboratory II directory has been created in your My Projects directory and that it contained a workspace information file named "Laboratory II.hw."
  - Double-clicked on the workspace file to re-launch DK.
- **Created a project:**
  - File→New. Created a new project "Counters." Added the project to the current workspace.
  - Deleted all build configurations but Debug. Copied the Debug configuration "Simulate", and deleted the Debug configuration.
- **Simulated a simple project:**
  - File→New. Created a new Handel-C source file "counters.hcc"

```
// counters.hcc
set clock = external "P1";
void main( void )
{ }
```

  - Simulated the project with "Simulate" build configuration selected. Verified that compilation produced no errors or warnings.
  - Modified the project by adding two global variables, an unsigned int with 3 bits and a signed int with 4 bits. Inside your *main()* code two parallel endless loops, one of which adds 1 to the unsigned variable, and the other of which adds one to the signed variable.
  - Built the simulation, and single stepped through it using the F11 key. Used the View→Debug Windows menu to be able to see the values of your two variables as you step through the code. Set a breakpoint in

the second while loop, and used F5 to set the simulation running. Used F11 to step from there.

- **Modified the code to implement each of the following changes, and simulated each one to make sure it works:**
  - Inserted a delay statement so that the unsigned variable increments half as often as the signed one.
  - Created an array of values, and wrote code that increments all of them in parallel.
  - Created a signed rom array containing an arbitrary list of signed 5-bit values, and wrote code that computes an 8-bit sum of the elements as fast as possible. Simulated the code to see how many clock cycles it took.
    - We used a standard library macro, *adjs()*, to convert a signed value of one width to another one. We need to put the statement #include <stdlib.hch> near the beginning of the program to use macros from the standard library.In addition, we provided the standard library for the linker.
  - Created a four-stage "pipeline" array with a width of 8. When the sum had been computed, pushed the value into the pipeline, and added the rom array to the sum again. Ensured that "pushing" onto the pipeline took only one clock cycle. Connected the output of the pipeline to an 8-bit variable named "display." Simulated the code to make sure it works.
- **Simulated and ran a PAL project:**
  - Created a new project called "Pipeline" in the Laboratory II workspace. Selected "Xilinx Virtex II" (the type of FPGA on the RC200E). Created two build configurations, a copy of Debug configuration called Simulate, and a copy of the EDIF configuration called RC200E. Deleted all the unused configurations from the project, leaving just the two new ones.
  - This project operated as follows: The rom, sum, pipeline, and display variables were the same as in the previous project. The contents of the display variable were to be displayed (in hexadecimal) in two seven segment displays. One pushbutton ("Add") was to be used to cause a new sum to be computed and pushed into the pipeline, and the other pushbutton ("Reset") was to be used to reset the circuit by zeroing the sum and making the pipeline "empty." Two LEDs were to indicate in binary how many stages of the pipeline are currently in use (0, 1, 2, and 3 or more). The seven segment displays were to be dark until values are received from the pipeline.
  - To use the PAL, included the header file *pal_master.hch*, and included *pal_master.hcl*, *pal_rc200e.hcl*, and *rc200e.hcl* in the Object/library modules list for the configuration.
  - Defined the preprocessor symbol PAL_TARGET_CLOCK_RATE before including *pal_master.hch* in the code.

- Ensured that the preprocessor symbol USE_SIM is listed in the Preprocessor tab of the Project Settings for the Simulate configuration and the symbol USE_RC200E is listed in the Preprocessor tab of the Project Settings for the RC200E configuration.
- Added the following three commands to the Build commands tab of the Project Settings for the RC200E configuration:
  - cd RC200E
  - call edifmake_rc200 EDIF
  - beep
- Entered a directory name in three places in the Project Settings for the RC200E directory: twice on the General tab, and in the Outputs view of the Build commands tab.
- Entered the part number of the FPGA in the Chip tab of the Project Settings for the RC200E configuration. It's XC2V1000-4FG456.

RESULTS:

- **Source code for the first simple project**:

```
// counters.hcc - Simple example.
/*****************************************************************
* Project   :    Counters                                         *
* Date      :    23 Feb 2004                                      *
* File      :    counters.hcc                                     *
* Author    :    Sergey Averchenkov                               *
*****************************************************************
* Desc      :    This program increments an array of values       *
*           :    by one in parallel.                              *
*****************************************************************/

#ifndef TRUE
  #define TRUE 1
#endif
#ifndef FALSE
  #define FALSE 0
#endif

set clock = external "P1";

int 4 arr[5]={1,2,3,4,5};



// main()
// -----------------------------------------------------------
/*
 *    The entry point of the program.
 */
void main(void)
{
  unsigned i;
```

```
  while(TRUE)
  {
    par(i=0; i < 5; i++)
    {
      arr[i]++;
    }
  }
}
```

- **Source code for the PAL project**:

```
// pipeline.hcc - PAL Project
/*************************************************************
* Project   :    Pipeline                                     *
* Date      :    23 Feb 2004                                  *
* File      :    pipeline.hcc                                 *
* Author    :    Sergey Averchenkov                           *
*************************************************************
* Desc      :    This project operates as follows:            *
*           :    The rom, sum, pipeline, and display variables  *
*           :    indicate data array, sum of elements in data   *
*           :    array, a stack of array sums, and the top value*
*           :    popped off the stack. The contents of the      *
*           :    display variable are be displayed (in hex)     *
*           :    in two seven segment displays. One pushbutton  *
*           :    (“Add”) is be used to cause a new sum to be    *
*           :    computed and pushed into the pipeline, and the *
*           :    other pushbutton (“Reset”) is be used to reset *
*           :    the circuit by zeroing the sum and making the  *
*           :    pipeline “empty.” Two LEDs indicate in binary  *
*           :    how many stages of the pipeline are currently  *
*           :    in use (0, 1, 2, and 3 or more). The seven     *
*           :    segment displays are dark until values are     *
*           :    received from the pipeline.                    *
*************************************************************/

#ifdef USE_SIM
#define PAL_TARGET_CLOCK_RATE 1000000
#else
#define PAL_TARGET_CLOCK_RATE 20000000
#endif

#include <pal_master.hch>
#include <stdlib.hch>

rom int  5 arr[5] = {1, 2, 3, 4, 5};
int      8 pipleline[4];
int      8 sum = 0;
int      8 display = 0;
unsigned 3 pipelineStages = 0;
```

```
unsigned 1 valSwitchAdd = 0;
unsigned 1 valSwitchReset = 0;

macro proc sleep(msec);

// main()
// ---------------------------------------------------------------
/*
 *    The entry point of the program.
 */
void main(void)
{
  unsigned 8 display2;
  unsigned 3 i;

  PalVersionRequire (1, 0);
  PalSevenSegRequire (1);
  PalLEDRequire(2);

  PalSevenSegWriteShape(PalSevenSegCT(0), 0);
  PalSevenSegWriteShape(PalSevenSegCT(1), 0);

  PalLEDWrite(PalLEDCT(1),pipelineStages[0:0]);
  PalLEDWrite(PalLEDCT(0),pipelineStages[1:1]);

  while(TRUE)
  {
    while(!valSwitchAdd)
    {
      PalSwitchRead(PalSwitchCT(0), &valSwitchAdd);
    }

    for(i=0; i < 5; i++)
    {
      sum += adjs(arr[i], 8);
    }

    par(i=0; i < 4; i++)
    {
      ifselect(i == 0)
      {
        pipleline[i] = sum;
      }
      else ifselect(i == 3)
      {
        display = pipleline[i-1];
      }
      else
      {
        pipleline[i] = pipleline[i-1];
      }
    } // END par
    display2 = (unsigned 8) display;
```

```
      if(pipelineStages == 3)
      {
        // Automatically enables SevenSeg
        PalSevenSegWriteDigit(PalSevenSegCT(0), display2[7:4], FALSE);
        PalSevenSegWriteDigit(PalSevenSegCT(1), display2[3:0], FALSE);
      }

      pipelineStages = (pipelineStages+1 > 3)? 3: (pipelineStages + 1);

      PalLEDWrite(PalLEDCT(1), pipelineStages[0:0]);
      PalLEDWrite(PalLEDCT(0), pipelineStages[1:1]);

      sleep(300);
      while(valSwitchAdd)
      {
        PalSwitchRead(PalSwitchCT(0), &valSwitchAdd);
      }
    } // END while
  } // END main

  // main()
  // -----------------------------------------------------------------
  /*
   *    Second thread: Reset functionality.
   */
  void main(void)
  {
    while(TRUE)
    {
      PalSwitchRead(PalSwitchCT(1), &valSwitchReset);
      if(valSwitchReset)
      {
        // Resets in 1 clock cycle
       par
       {
          // Reset sum
          sum = 0;

          // Reset pipleline counter
          pipelineStages = 0;

          // Reset pipeline
          par(i=0; i < 4; i++)
          {
            pipleline[i] = 0;
          }

          PalSevenSegWriteShape(PalSevenSegCT(0), 0);
          PalSevenSegWriteShape(PalSevenSegCT(1), 0);

          PalLEDWrite(PalLEDCT(1), 0);
          PalLEDWrite(PalLEDCT(0), 0);
        }
      }
```

```
  }
}

// sleep(msec)
// ------------------------------------------------------------
/*
 *    Suspends the execution for a number of milliseconds.
 */
static macro proc sleep(msec)
{
  macro expr Cycles = (PAL_ACTUAL_CLOCK_RATE * msec) / 1000;
  unsigned (log2ceil (Cycles)) Count;

  Count = 0;
  do
  {
      Count++;
  }
  while (Count != Cycles - 1);
}
```

DISCUSSION:

One of the problems we encountered was the ability to correctly and precisely read switches on the device. In software simulation when user clicks a button switch, the "push" event occurs only once. In hardware implementation, however, metal contacts that represent a switch can accidentally bounce off while user pushes the button thus resulting in several "push" events occurring during a very short time period (several times a second). Such behavior can produce very unpredictable or undesirable results and make it difficult for the user to provide correct input for the device.

It would be possible to seek a more reliable switch, but we were limited by the available hardware in RC-200E unit and were forced to find a solution using Handel-C language. The solution was found and is presented in the Results section of this report.

Another issue relates to concurrency. In this application we have two concurrent and independent threads running. First one monitors the Add switch and performs all calculations specified by the program's behavior, and the other monitors the Reset switch and resets all values in a single clock cycle. Add feature works only once per switch activation, while Reset initializes the variables continuously while the switch is being held in the On position. Even though these two threads access (read and write) the same set of variables, no synchronization was deemed necessary. The only time when data collisions may occur is when user presses the Add and Reset at the same time. Due to the fact that Reset operates very fast, any possible data collisions that may occur within the current clock cycle will be corrected during the subsequent one. The case when user holds Reset switch for exactly one clock cycle is highly improbable because of the device's internal frequency of 50Mhz. This would mean that the user would have to press and

release the Reset in exactly 0.00000002 seconds. Physical capabilities of the switch and the average user would not allow this level of precision.