University of Pennsylvania
Department of Electrical and System Engineering
System-on-a-Chip Architecture

ESE532, Spring 2017 HW4: Thread Parallel Wednesday, February 1

Due: Friday, February 10, 5:00PM

In this assignment, we will map the MPEG-2 encoder implementation on multiple MicroBlaze processor cores. We will start with the same encoder source as in homework 3.

Collaboration

Work with assigned partners.[1] Writeups should be individual. Partners may share code and profiling results and discuss analysis, but each writeup should be prepared independently.

Hardware Platform

So far, we have used the provided `zed` platform that instantiates only the hard-wired functionality of the Zynq SoC, such as the ARM processors. Xilinx calls this part of the SoC the Processing System (PS). In addition to the PS, the provided platform configures 12 MicroBlaze processors on the programmable Logic (PL). The Xilinx MicroBlaze is a soft-core microprocessor. "Soft-core" refers to the fact that the processor is implemented on a reprogrammable device, as opposed to a "hard-core" processor, which is physically implemented in silicon (like the ARM processors on the Zynq). The processor is configurable, i.e., one can select parameters such as the instruction and data cache sizes and the inclusion of multipliers, dividers, etc. Configuration happens at design time, before the processor is mapped on the FPGA. For this assignment, you will be treating the processor array as fixed hardware (it represents a particular, fixed SoC platform); for future assignments (at least for the project), you will be able to customize the cores and other logic.

Memory Map

Every instruction or data bus of each processor has its own view of the memory hierarchy, which is represented by an address map. For every application (process running on one of the processor cores), we dedicated an area in memory; that is, we are effectively giving each processor a private memory to store the instructions for the process it runs; we're just

[1] To be posted under Files in canvas.

putting them in a single shared address space. The following table describes the SDRAM memory allocation as seen from the data bus of the processors:

Start address	End address	Description
0x00100000	0x03FFFFFF	ARM Application (only visible to ARM)
0x10000000	0x10FFFFFF	MicroBlaze 0 application
0x11000000	0x11FFFFFF	MicroBlaze 1 application
0x12000000	0x12FFFFFF	MicroBlaze 2 application
0x13000000	0x13FFFFFF	MicroBlaze 3 application
0x14000000	0x14FFFFFF	MicroBlaze 4 application
0x15000000	0x15FFFFFF	MicroBlaze 5 application
0x16000000	0x16FFFFFF	MicroBlaze 6 application
0x17000000	0x17FFFFFF	MicroBlaze 7 application
0x18000000	0x18FFFFFF	MicroBlaze 8 application
0x19000000	0x19FFFFFF	MicroBlaze 9 application
0x1A000000	0x1AFFFFFF	MicroBlaze 10 application
0x1B000000	0x1BFFFFFF	MicroBlaze 11 application

In addition, we devote areas for communication between the ARM and the MicroBlaze (and potentially among the MicroBlaze processors). We put those in a 256 KB On-Chip-Memory (OCM). The following table shows the memory allocation of the OCM:

Start address	End address	Description
0xFFFC0000	0xFFFC3FFF	Communication area 0
0xFFFC4000	0xFFFC7FFF	Communication area 1
0xFFFC8000	0xFFFCBFFF	Communication area 2
0xFFFCC000	0xFFFCFFFF	Communication area 3
0xFFFD0000	0xFFFD3FFF	Communication area 4
0xFFFD4000	0xFFFD7FFF	Communication area 5
0xFFFD8000	0xFFFDBFFF	Communication area 6
0xFFFDC000	0xFFFDFFFF	Communication area 7
0xFFFE0000	0xFFFE3FFF	Communication area 8
0xFFFE4000	0xFFFE7FFF	Communication area 9
0xFFFE8000	0xFFFEBFFF	Communication area 10
0xFFFEC000	0xFFFEFFFF	Communication area 11

The OCM memory is not cached and is therefore guaranteed to be consistent among the processors. That is why it can be used safely for processor-to-processor communication.

The SDRAM data is cacheable in the MicroBlaze processors and therefore **not** kept consistent among the processors. This will work fine as long as you use it for private or read-only data from the MicroBlaze processor threads. However, if you try to write to this memory from one processor and read it from another, there is no guarantee the reader will see the new value written. If you follow the discipline of privatizing all data in SDRAM and only using the OCM for communications, you will not have to think directly about the impact of the caches hiding data changes in the SDRAM region.

Nonetheless, it may be useful to store large shared data in the SDRAM that remains read-only during most of your computations, but needs to change periodically—such as the current

frame that you are encoding. This is reasonable to do, but you will then need to take care to flush the MicroBlaze data caches when this shared data changes. You can cause the MicroBlance to force a data cache flush with: `Xil_DCacheFlush()`. Note that the visible range of SDRAM of the MicroBlaze processors is limited to 0x10000000–0x1FFFFFFF, so the MicroBlazes cannot access the ARM application memory area.

Exploring the Platform

The hardware platform was created in Vivado. Vivado is a Xilinx tool for developing hardware for the FPGA. We will only explore the platform in this assignment. It should not be necessary to make changes.

1. Download the hardware platform from the course website and unpack it.
2. Launch Vivado. In Linux, you can launch Vivado from the terminal using the command `vivado`.
3. Open the project with the platform. Select *File→Open Project...* from the menu. In the *Open Project* dialog, choose the `hw4_pfm/vivado/hw4_pfm.xpr` file in the platform that you extracted before.
4. Further instructions for exploring the platform are given in the *Homework Submission* section.

Building a Multithreaded Example Design

In this assignment, we will create many projects, so we recommend that you use recognizable names. Note that our method to execute code on the 12 processors is rather laborious and primitive, but we don't have a better method at this point.

1. Launch SDSoC, and select a workspace. We will create many projects for this assignment, so you may want to create a new workspace.
2. Create a *Hardware Platform Specification* for the downloaded platform. We suggest calling it `hw4_hw`. Select the `hw4_pfm/vivado/hw4_pfm.sdk/hw4_top_wrapper.hdf` file from the downloaded platform as target.
3. Create an empty *Application Project* for a standalone OS on the `ps7_cortexa9_0` processor, which is one of the ARM cores. We will not be using the second ARM core. We suggest calling the project `hw4_arm_app`. In the *New Project* dialog, choose the option to create a new BSP. We suggest that you call it `hw4_arm_bsp`. Enable the `xilffs` library in the BSP because we will load the applications from the SD-card in this exercise.

4. Create 12 empty *Application Project*s for a standalone OS. Target each of the projects to a different MicroBlaze processor. These processors are called `microblaze_0`, `microblaze_1`, etc. We suggest naming the projects `hw4_mb0_app`, `hw4_mb1_app`, etc. Select the option to create a new BSP in the *New Project* dialog of each project that you create. We suggest naming the BSPs `hw4_mb0_bsp`, `hw4_mb1_bsp`, etc.

5. Create a First Stage Boot Loader (FSBL). We will load the applications automatically from the SD-card. The FSBL is responsible for this. To create an FSBL, create a *Application Project* for a standalone OS. We suggest to call this project `hw4_fsbl`. In the *New Project* dialog, choose the option to use an existing BSP. The FSBL will run on the ARM. We already have a BSP for the ARM, `hw4_arm_bsp`, so reuse it. Choose the *Zynq FSBL* as template on the next page and finish the dialog.

6. Increase the number of partitions that FSBL can load. By default, the limit is 14 partitions. Partitions are bitstreams, code or data segments from the application, or data sets. An application typically has multiple partitions, so 14 partitions are not sufficient for 12 applications. You can change the limit by modifying the value of the `#define MAX_PARTITION_NUMBER` in the header file `src/image_mover.h` of the FSBL project.

7. Pass the symbols `NON_PS_INSTANTIATED_BITSTREAM` and `DEBUG_FSBL` to the compiler while building the FSBL. The first symbol guarantees that the MicroBlazes do not come out of reset before the ARM has initialized the necessary data. `DEBUG_FSBL` will make the FSBL generate debug output, which may come in handy if your application is not loaded correctly. To pass the symbols, open the *Project Properties* of the FSBL project. Select *C/C++ Build→Settings* on the left side. In the *Tool Settings* tab, choose *ARM v7 gcc compiler→Symbols*. Under *Defined symbols (-D)*, add the symbols. Note that this has the same effect as adding a `#define` with the symbol on top of every C source file in the project, but you won't have to change the source.

8. Create a source file in the `src` directory of your ARM application project, and copy the code from `hw4_arm_app.c` in the `hw4_example.tar.gz` archive. You can download the archive from here.

9. We have to link the `ps7_init.c` into the ARM application. Choose the `src` directory of the ARM application project project. Right-click on it, and select *Import...*. In the dialog box, select *General→File System* and press *Next*. Browse to the hardware platform specification workspace. Choose the `ps7_init.c` file. Reveal the link options by pressing the *Advanced...* button. Enable *Create links in workspace* and dismiss the dialog.

10. Add the `hw4_hw` directory to the includes of the ARM application.

11. Create a source file in the `src` directory of each of the MicroBlaze application projects with the code from `hw4_mb_app.c` in the `hw4_example.tar.gz` archive.

12. Replace `<ADDRESS>` and `<NUMBER>` in each of the MicroBlaze application source files with the address of the communication area in the table above and the number of the MicroBlaze respectively.

13. Make sure that the MicroBlaze applications are loaded into the main SDRAM. The instantiated memories in the PL have a lower latency, but they are too small to store the code. Select a MicroBlaze project. Go to *Xilinx Tools→Generate linker script* in the menu. Change the setting of *Place Code Sections in:* to `ps7_ddr_0_HP0_AXI_BASENAME`. Repeat for the data and heap and stack sections. Generate the linker script by pressing *Generate*. Repeat this for all MicroBlaze application projects. Don't do this for other projects.

14. Build all projects.

15. Create a file called `hw4.bif` with the following lines:

```
the_ROM_image:
{
        [bootloader]<Workspace>/hw4_fsbl/Debug/hw4_fsbl.elf
        <Workspace>/hw4_hw/hw4_top_wrapper.bit
        <Workspace>/hw4_arm_app/Debug/hw4_arm_app.elf
        <Workspace>/hw4_mb0_app/Debug/hw4_mb0_app.elf
        <Workspace>/hw4_mb1_app/Debug/hw4_mb1_app.elf
        <Workspace>/hw4_mb2_app/Debug/hw4_mb2_app.elf
        <Workspace>/hw4_mb3_app/Debug/hw4_mb3_app.elf
        <Workspace>/hw4_mb4_app/Debug/hw4_mb4_app.elf
        <Workspace>/hw4_mb5_app/Debug/hw4_mb5_app.elf
        <Workspace>/hw4_mb6_app/Debug/hw4_mb6_app.elf
        <Workspace>/hw4_mb7_app/Debug/hw4_mb7_app.elf
        <Workspace>/hw4_mb8_app/Debug/hw4_mb8_app.elf
        <Workspace>/hw4_mb9_app/Debug/hw4_mb9_app.elf
        <Workspace>/hw4_mb10_app/Debug/hw4_mb10_app.elf
        <Workspace>/hw4_mb11_app/Debug/hw4_mb11_app.elf
}
```

    Replace all strings in angle brackets with the locations on your system. If you chose different project names than we suggested, you will have to adapt those too. You can make the file with an ordinary text editor in a convenient location. Note that you can also create the file using the SDSoC GUI by choosing *Xilinx Tools→Create Boot Image* from the menu.

16. Create the boot image by issuing the following command in the directory with the `hw4.bif` file:

```
bootgen -image hw4.bif -o boot.bin -w
```

The `-w` flag orders `bootgen` to overwrite an existing `boot.bin` file. Note that you can skip this step if you used the GUI in the previous step.

Debugging a Multithreaded Example Design

1. Place an SD-card in the memory card reader of your PC with SDSoC.
2. Copy the `boot.bin` file to the root directory of the SD-card.
3. Make sure that your ZedBoard is shut down.
4. Make sure that your PC has finished writing to the SD-card, and place the SD-card in the card slot on the ZedBoard.
5. Change the boot mode of the ZedBoard to SD-card. You can do that using the same jumpers as in homework 1. Refer to the ZedBoard manual for the right positions.
6. Connect both USB connectors to the PC as before.
7. Power up the ZedBoard. The blue `DONE` LED should light up after a while.
8. Open the *Debug Configurations* dialog in SDSoC. Select *Xilinx C/C++ application* on the left side. Create a new configuration by pressing the corresponding icon above the same list. In the *Target Setup* tab, change the *Debug Type* to *Attach to running target*. Press *Debug* to launch a debug session.
9. In the *Debug* perspective, you will see the name of your debug configuration in the *Debug* window. If you expand the tree further, you will see the ARM cores under *APU*, and the MicroBlaze cores under *xc7z020→Debug Module at USER2*. You can select any of the cores and press the suspend icon above the window to stop it temporarily. The current instruction will be shown in a *Disassembly* window. Most likely, this will be of little use because the machine code is not annotated with source lines.
10. We can show the source lines by adding symbol files to the debug configuration. A symbol file contains the addresses of source lines and statically allocated data objects in your code. They are generated for debug purposes and typically not provided with the final application. The default compiler configuration does not generate symbol files, but it puts the symbols in the executable (ELF) file. Open the *Debug Configurations* dialog again. Select the debug configuration that you created and go to the *Symbol Files* tab. Change the *Debug context* to the processor for which you would like to provide the symbols. Press *Add...* on the right side and enter the path to the ELF file for the given processor. It should be in the `Debug` directory of the associated application project. Close the dialog by pressing `OK`. You can add the symbol files for all processors, but it is probably wiser to add them as you need them.
11. Once you have added a symbol file, you can debug code on the processor as usual.

Homework Submission

1. **Learning and navigating the tools.**

   Document all the problems you encountered in working the assignment and how you solved them. For each, include:

   (a) Step (if applicable)

   (b) OS you were running

   (c) Error message or unexpected behavior you encounted

   (d) How the problem was resolved

   (e) How you found the solution (e.g., how did you experiment or reason through an answer, which document had the answer, where on the web did you find an answer, which student or TA was able to point out the answer)

2. **Teamwork**

   (a) Document the work sessions with your partner (where, when, duration).

   (b) How did you collaborate on the work? (pair programming is acceptable, explain)

   (c) What background basics, tricks, or techniques did you learn from your partner?

   (d) Provide highlights for cases where working together allowed you to understand deeper, combine insights, avoid pitfalls, and/or build confidence in your directions and solutions.

3. **Hardware Platform**

   (a) Open the *Block Diagram* of the platform in Vivado. In the *Flow Navigator* on the left side, press *Open Block Design*. Zoom buttons are on the left side of the diagram. Thick lines connecting blocks are buses; thin lines are single wires. Select a block to see information in the *Block Properties* window. Double-click on a block to see the most important settings in a dialog. Answer the following questions about the current configuration:

      i. How large is the instruction cache of each MicroBlaze?

      ii. What is the widest multiplication that the MicroBlaze can perform?

      iii. How many BRAMs does each MicroBlaze use?

      iv. Describe how `microblaze_3` can access data in the SDRAM. Mention the buses and components that are involved.

      v. What is the clock frequency of the MicroBlaze?

      vi. At which address does `microblaze_9` start after booting up? You can find it in the *Properties* tab of the *Block Properties* window. The address is given under `CONFIG`→`C_BASE_VECTORS`.

(b) Open the *Address Editor*. You can find it among the tabs immediately above the block diagram. The editor shows a memory map for every instruction and every data bus on each MicroBlaze processor. No address map for the PS is shown because it is not configurable.

i. How large is the memory area reserved for SDRAM in the memory map of the data bus of `microblaze_1`?

ii. Which processors can see the instruction at the starting address of `microblaze_9`?

(c) Look at the *Project Summary*. You can open it by selecting *Window→Project Summary* from the menu. Which resource type has the relatively fewest resources remaining in this implementation? What is the purpose of this type of resource?

4. **Multithreading**

Accelerate the MPEG2 encoder by introducing threads. You can use the example as a starting point. You may change the format of the `communication_area` structure as you wish.

(a) Revise the provided code to run `fullsearch` on one MicroBlaze

i. Report the encoder execution time when `fullsearch` is executed on one MicroBlaze.

ii. Describe any debugging you needed to do to get the `fullsearch` thread working correctly on the MicroBlaze

(b) Consider how to decompose `fullsearch` into 2 threads

i. Describe 3 possible decompositions.

ii. Give the advantages and disadvantages of each decomposition.

(c) Select and implement one solution

i. Include the code for the 2 threads in your writeup. This should just be the parts of the code that you changed from the original.

ii. Report the speedup that running this decomposition on MicroBlazes achieves.

iii. Describe any additional debugging you needed to do to get the `fullsearch` thread working correctly on two MicroBlaze cores.

iv. Describe any optimizations and tuning you needed in order to achieve this speedup.

(d) Scale to all 12 MicroBlaze processors

i. Describe a promising encoder decomposition that makes use of all processors.

ii. What speedup do you expect under ideal circumstances?

iii. Include the code for your 12 thread solution. Again, just the code changed for this solution.

iv. Report the speedup that running this decomposition achieves.

v. Describe debugging, optimization, and tuning to achieve correct operation and speedup.

vi. Explain potential reasons for the difference between ideal and actual speedup (identify non-computational bottlenecks and overheads (computational or not) that are impacting the execution).