Porting GNU Radio to Multicore DSP+ARM System-on-Chip – A Purely Open-Source Approach

Shenghou Ma*, Vuk Marojevic*, Philip Balister†, and Jeffrey H. Reed*
*Bradley Dept. of Electrical and Computer Engineering
Virginia Polytechnic Institute and State University
Blacksburg, Virginia, USA
{minux,maroje,reedjh}@vt.edu
†Open SDR
PO Box 11789
Blacksburg, Virginia, USA
philip@opensdr.com

***Abstract*—GNU Radio is a versatile fast-prototyping platform for Software-Defined Radio (SDR). This paper describes our mission of porting GNU Radio to Texas Instruments' KeyStone II Multicore DSP+ARM System-on-Chip (SoC). We describe our long term vision providing unified scheduling over the heterogeneous multiprocessor SoC and initial results on this revolutionary path.**

***Index Terms*—GNU Radio, System-on-Chip, Open-Source.**

I. Introduction

SOFTWARE-defined radio (SDR) is becoming omnipresent both in academia and industry. Vanu, for instance, is deploying SDR base stations for a decade already. Amarisoft has recently implemented an LTE base station, or eNode-B, purely in software that runs on powerful general-purpose processor (GPP). GNU Radio is gaining momentum in education and research. It has a growing user community, which provides support through their mailing list, and organizes regular phone conferences and meetings at diverse conferences and events. GNU Radio is open-source and provides a framework for rapid waveform development and testing, supporting a variety of popular radio front ends. It is GPP-centric and is only recently being ported to other devices, such as graphics-processing units (GPUs) [1].

Systems-on-chip (SoCs) provide important advantages over GPPs for building wireless communication systems: processing capacity, system integrity and power efficiency, to name a few. Heterogeneous Multiprocessor-SoCs (HMP-SoC) are being deployed not only in radio access network element, but also in mobile devices [2]. However, developing applications (waveforms) for HMP-SoCs is a much more tedious task than for GPPs. The designer needs to deal with heterogeneous devices and manage the data flows among them as sophisticated development tools and schedulers are not commonly available. Several vendors are developing powerful HMP-SoC with the goal of deploying cheap and versatile LTE transceivers.

Our goal is to provide a fully open-source approach for SDR education and professional development on a modern SoC. This paper describes our vision for the Keystone II Multicore DSP+ARM SoC from Texas Instruments to be used in SDR education, research and development. Porting GNU Radio to the ARM allows directly interfacing with radio frequency (RF) front ends. This provides the excitement of over-the-air transmissions for students and researchers. Having GNU Radio running on the ARMs can be useful for DSP developers as well if their algorithms, running on a DSP core, can be integrated into a GNU Radio flow graph. We will evaluate the different approaches and justify a purely open-source approach.

II. Context

GNU Radio provides an open-source framework for rapid waveform design and prototyping. It provides a number of readily available signal processing blocks and also easily integrates third-party blocks. The waveform designer can glue together these blocks to create an intended data flow graph.

Aided by the availability of compatible RF front-end hardware, such as the Ettus Research's Universal Software Radio Peripheral (USRP) series. GNU Radio is ideal for education and research. However, GNU Radio

is not without its limitations: Designed for PCs, GNU radio is not easily portable to non-GPP-based processing platforms.

The trend in computing is towards heterogeneous multiprocessing, where different instruction set architectures (ISAs) and micro-architectures are integrated in a single system. This trend is observable by looking at the current top500[1] supercomputer list. The fastest supercomputer in the world in January 2014 is the Tianhe-2 (MilkyWay-2)[2] in the National Super Computer Center in Guangzhou, China. It uses both the Intel Xeon E5-2692 12-Core general purpose processor and Intel Xeon Phi 31S1P many-core processor and has a total of 3120000 computing cores and 1 PB memory. Using 17.808 MW electricity, it can provide a theoretical peak computation throughput of 54.902 Peta-Flops per second. The second ranked is the Titan, which is a Cray XK7. It uses AMD Opteron 6274 16-Core processors and Nvidia K20x GPUs to provide a theoretical peak computation throughput of 27.112 Peta-Flops per second. Such heterogeneous multiprocessing architectures provide a higher processing integration and power efficiency than equivalent GPP-based solutions.

SDR requires a huge amount of computational resources, because of the high date rate and sophisticated signal processing algorithms. In this aspect, it looks very much like a supercomputer at a smaller scale. But, you cannot afford to carry around a mega-watt smartphone. To be practical, SDR platforms also need the virtue of power efficiency. This is where GPPs fail to deliver their promises. Nowadays, almost all smartphones are using the ARM processor precisely because it can deliver the ever-increasing requirement of computation at affordable power levels. However, using ARM processors alone cannot provide the necessary processing power. Digital signal processors (DSP), which are optimized for multiply-accumulate (MAC) operations close this gap.

A DSP is the choice for implementing typical digital signal processing algorithms in software, but it has the disadvantage of not being very easy to program (partly because of the lack of smart compilers that do not need manual hints in order to utilize the full potential of the chip). ARMs and DSPs are therefore combined to take advantage of both processor types, the processing power of the DSP and the flexibility of the ARM. Texas Instrument's has recently released their Keystone II multicore DSP+ARM SoC.

[1]http://www.top500.org/lists/

[2]http://www.top500.org/system/177999

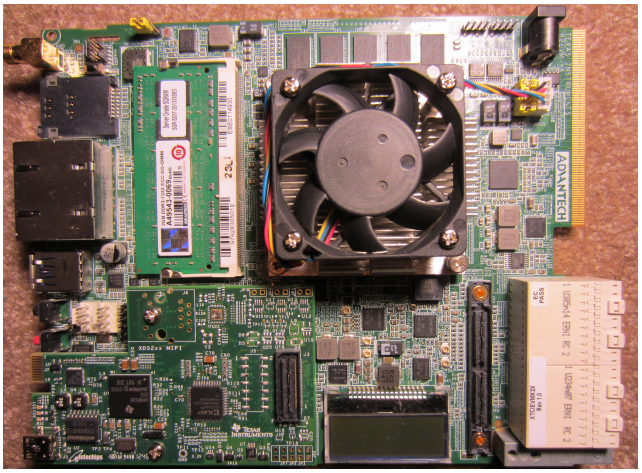

Fig. 1. The Target EVMK2HX Keystone-II Evaluation Board [4]

A. The Open-Source Aspect

GNU Radio is an open-source project distributed under the General Public License. This calls for open-source SDR tools for integrating GNU Radio flow graphs with processing block running on the DSP cores of the Keystone II or similar HMP-SoCs. The reasons for this are manifold.

Firstly, open-source software is generally more accessible, so relying on open-source technology makes the result easily repeatable. Secondly, having the source code available means that one does not need to rely on vendor for any bug fixes and feature enhancements, and this also provides long-term stability against vendor dropping support for the software. This is especially a problem in the SoC market where the fast proliferation of new devices lead to limited-time support for older generations. Last but not the least, open-source means one can modify the software to fit his or her own needs, and more importantly, to share freely the modification. Although some proprietary software comes with source code, the license agreement generally forbid the user from distributing modified copies. This limits the ability of sharing among the research and education communities.

B. Hardware Platform

Fig. 1 shows the EVMK2HX evaluation board, produced by ADVANTECH [3], used in this paper. Under the fan and heat sink lies the main multi-core DSP+ARM SoC TCI6638K2K from Texas Instruments.

Fig. 2 depicts the functional block diagram of the TCI6638K2K chip. The chip contains a number of processing cores, accelerators and peripherals [5].

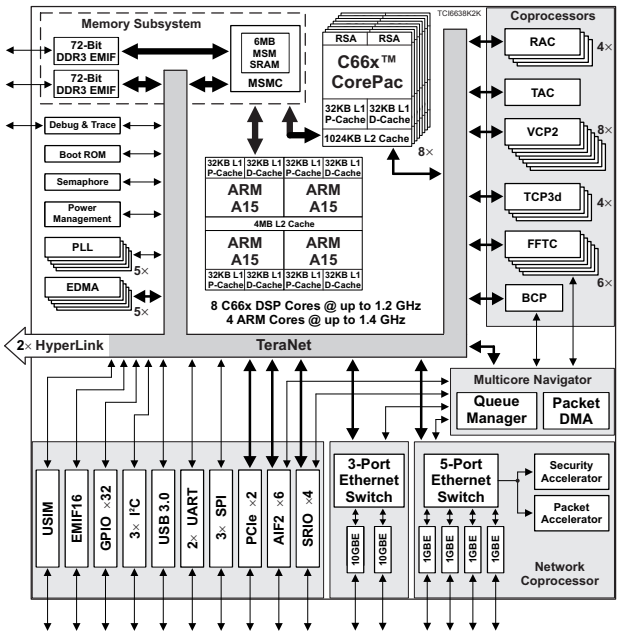

Fig. 2. The Diagram of TCI6638K2K [3]

The chip features eight TMS320C66x fixed/floating point DSP core running at 1.2GHz, each capable of providing:

- 38.4 GMACs for fixed point,
- 19.2 GFlops for single precision floating point.

Each DSP core also contains 32 KB of L1P cache, 32 KB of L1D cache and 1MB of local L2 cache. All of the L1P/L1D/L2 can be separately configured as SRAM or cache, or a combination of them. Also, these caches are accessible from the 40-bit physical memory address space, so that the ARM cores (or even other DSP cores) can access other DSP's cache/local SRAM memory, which naturally provides a way to share information between the cores. Note that the L2 cache/SRAM is big even for PC standards. One can thus load the entire DSP program and all its required data into L2 RAM and thus bypass the main chip memory matrix, reducing both access latency and the contention to the chip-wide memory matrix.

Four ARM Cortex-A15 MPCores at up to 1.4GHz. They serve as the main controlling cores of the chip. ARM Cortex-A15 is the single most powerful 32-bit ARM implementation to date. Four of them have ample amount of computing resources for running Linux. Each ARM has 32 KB L1I and L1D caches and shared access to 4 MB of L2 cache.

The chip also includes a Multicore Shared Memory Controller (MSMC) which provides another 6MB of SRAM shared by all 12 cores. It also has memory protection capability so that we can limit each core's access privilege to this memory for safety and isolation.

The SoC also has a lot of hardware coprocessors supporting typical digital signal processing functionalities of modern base stations [3].

Since the SoC contains 12 heterogeneous cores, communication and debugging of multi-core application poses a bigger challenge than for traditional SoCs or GPPs with a smaller core count. In order to facilitate inter-core communication and synchronization, the SoC also provides a Multicore Navigator.

C. Related Work

In [1] and [6], the authors managed to use GPGPU (more specifically, NVidia GPGPU with CUDA) to accelerate certain GNU Radio blocks. They rewrote the blocks and rewrote the inner loops in CUDA compute kernels and devised a way to call those computer kernels. While this project sucessfully brings GPU-accelerated blocks into the equation, it requires rewriting the critical portion of the blocks to CUDA computer kernels, which is a challenge due to way GPU works. Nvidia GPUs contain a lot of processing units with very large register sets and very fast and fine grained thread switching hardware support. Every core is executing the same kernel on a GPU. Because the bottleneck is the memory access, whenever a thread requires access to shared memory (each core also has local memory, which is not subject to this limitation) and is stalled because the contention on the GPU memory bandwidth, the hardware thread scheduling unit will immediate switch to another thread. This means that if there are enough runnable threads available, the memory access latency, although very big and on the scale of thousands of core clock cycles, can be completely hidden because the processing unit always has new work to do. To fully exploit this architecture, the programmer will need to partition the program into many smaller pieces and do some coordination on memory accessing pattern to optimize the usage of the special memory controller inside the chip (for example, the memory controller can do memory write aggregation; because the GPU memory bus is very wide, some more than 1024-bit, it is very critical to fully use this wide bus). This paradigm is particularly suited to solving image processing problems, where the program can be naturally divided into small sub-blocks and assign the processing of each sub-block to a different thread. However, for SDR applications, this might not be the best fit. Because we are essentially

doing 1D processing, and as the window length is usually limited (to prevent large latency), there might not be enough parallelism available to fully exploit the GPU architecture.

In [7], the authors are using TMS320C6670 connected through PCIe to the host PC to do some computation offload, and they created special DSP blocks in GNU Radio. This work is very similar to what we propose, except the way the DSP is connected and how the control of DSP is handled. As the DSP is connected to host PC through PCIe point-to-point protocol, and does not share memory with the host, there is problem in that excessive data movement could eliminate any DSP computation performance benefits. As we presented in the previous subsection, this project utilizes a multi-core SoC that have a large shared memory on chip, so that we can eliminate the cost of data copying between host processors (ARM) and DSP accelerators (DSP). It worthy to mention that [7] also uses specially created DSP blocks in GNU Radio just like what we are proposing.

In [8] the authors introduced an interesting work (GReasy) to get GNU Radio blocks running in FPGA without the lengthy compilation time typically associated with FPGA designs. Although not directly related to this work, their approach of pre-compiling the FPGA blocks and assembling the blocks on the fly is still fascinating, and fits within our long-term vision. One limitation is that [8] does not address any scheduling problem associated with GNU Radio, and instead they assemble the complete flow graph inside FPGA. Although FPGA is generally suited to digital signal processing algorithms involved in SDR, using FPGA to implement signal processing blocks typically involves much more development effort than other purely-software based approaches.

III. Approach

In this section we will outline our long-term vision and the evolutionary path to reach it.

A. Long-term vision

The future of GNU Radio is clearly multi-core, and more specifically, as mentioned in the preceding sections, heterogeneous architectures. We can imagine that once GNU Radio is ported to non-GPP architectures, it will allow developing an unified scheduler capable of scheduling to multiple heterogeneous cores (including, but not limited to, networked clusters, GPUs, DSPs, custom accelerators like the Parallella project and even FPGAs [8]).

Imagine that after you draw your data flow graph, click Run, GNU Radio automatically compiles the code for each block and distribute memory intensive application to GPU, bitwise and interger-intensive blocks to FPGA and even to remote server cluster, all without human intervention. That should be the picture for tomorrow's GNU Radio optimized on heterogeneous computing architectures.

To achieve that goal, GNU Radio will need an unified scheduler that can schedule blocks to execute on the processor (or worker in case of remote server cluster) that is the most fit and optimizes the overall latency and other costs (e.g. power consumption).

The unified scheduler will need to take all these parameters into account:

- Proper way to share the data; should it stream the data via message passing channels or is shared memory a better approach? It will also need to take care of non-local memory (NUMA) issues.
- Throughput of the block running on a particular processor (worker).
- Latency of using different processors.
- Amount of data that needs to be transferred in real-time and the communication throughput and latency.
- Processor setup overhead. (For example, FPGAs need some time to setup. Please refer to [8] for an interesting approach to reduce such setup time.)

Because GNU Radio blocks differ greatly in their processing needs, it makes sense not restricting the scheduler to merely schedule blocks to processors (workers). It is also interesting to investigate how the scheduler could combine adjacent blocks or disassemble huge blocks on demand to suit the given processor and processor availability.

Also, for non-local processors connected by low throughput networks, perhaps it is beneficial to automatically insert fast stream compressors and decompressors transparently into the chain. Although this increases the computation demands of the involved blocks, the decrease in amount of data to be transfered and the associated communication latency reduction might be worth the additional computing effort.

In order for this to work, all processing blocks need to be able to run on all supported processors. One naïve way might be to implement every block on every supported processor, but this does not scale very well, as for N blocks and P supported processors, you will have to write $N \times P$ different implementations. And worse yet, every additional block involves implementing it on all P

processors, and every newly supported processor architecture involves rewriting every one of the N blocks. In order to achieve better scalability, and ease of adding new blocks and new processors, we need to find a common implementation language so that each processor supports that language reasonably efficiently.

Of course, this approach will not exploit the full potential of each supported processors, so for specific blocks, and likely for the processor that most efficiently execute such block, one can write hand-optimized implementation.

Also, to relieve the waveform developers from inter-block communication requirements, we should strive to decouple the core processing algorithm from the code needed to interfacing with the other blocks. We will also need to provide smarter memory management intrinsics for non-local memory management.

But in which language should we write the core processing algorithm? The answer to this would make an important research contribution that will not only benefit the GNU Radio community, but also the whole heterogeneous multiprocessing community. For this paper, we simply pick a subset of the venerable C programming language for that job. The reason is that the C programming language is simple and universal enough that most current processors have directly support for it. Even for the more data-flow oriented and traditionally hostile-to-C FPGA targets, people are actively working on high-level synthesis tool that could generate low-level hardware description language (HDL) code directly from algorithmic C functions. So picking a restricted subset of C seems a good choice for now. It is conceivable that future advances in data flow languages might provide better alternatives that suit the available computing technologies.

B. Evolutionary Path

We need to take smaller steps to achieve the ambitious goal outlined in the preceding section. We propose dividing them into the following three milestones.

First, support to run remote blocks on the same architecture as the host machine. This involves manually selecting parts of the flow graph and assign them to another machine for execution. GNU Radio first partitions the flow graph into sub-flow graphs accordingly, and inserts required TCP sources and sinks for each inter-connections between the sub-flow graphs. Each sub-flow graph is treated as if it were self-contained, generating a Python script for each one of them to be sent to the remote machine for execution. The major work involved is mainly on the GUI for enabling the partition, automatic TCP source/sink insertion and managing the communication between machines. There will not be any scheduler changes, each worker machine essentially runs their own separate copy of scheduler, and there is no global coordination between them.

Second, extending support to DSP, GPU and other processors, but still rely on manual partitioning and assigning the blocks to each processor. This is the most tedious step, because it may require rewriting each block. The major work is separating the core processing code and write it in a portable language of choice (for example, OpenCL kernels, or simple C subsets), and also extend the communication framework developed in the preceding milestone to shared memory and other point-to-point links (e.g. PCIe). At this stage the scheduling of blocks is still local, and there is no global coordination. This step might be tedious, but it is considered feasible.

The third step is designing, modeling and developing the unified scheduler. This is the major research challenge and remains for future research.

This paper focuses on the second milestone for the C66x DSP target.

C. Specific Issues for this Target

As in other multi-core programming project, managing the data flow or memory hierarchy is the single most important factor for performance. Since this work focuses on offloading computation to DSP cores (part of the second milestone aforementioned), the following sections will generally ignore the ARM cores running the Linux operating system kernel.

As discussed in section II, the DSP cores in TCI6638K2K SoC use a complex 3-level memory hierarchy. Firstly each core has 32 KB of Level 1 Program and Data caches, both of them could be configured into a flexible split of cache and SRAM. Each core also has 1MB of private Level 2 cache, which can also be configured as SRAM.

Based on the timing data from [9], we conclude that setting L2 to SRAM will reduce L1 cache miss stall time. As L1D and L1P are only 32 KB in size, generally not big enough to hold all the data and program, we will leave it as cache (if the program does fit in 32 KB, then we can use L1P as program memory, and it will provide the best performance.)

Fig. 3 shows the proposed data flow between ARM and DSP cores. Normally the DSP cores are kept in reset. When GNU Radio starts executing a DSP block, the code for the DSP is compiled on the fly and loaded to

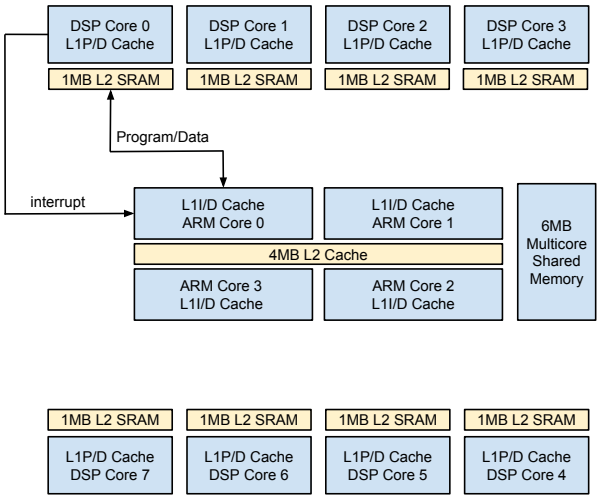

Fig. 3. Data Flows Between ARM and DSP Cores

the available DSP core's L2 SRAM along with the data. Then the ARM will release the target DSP core from reset and wait for its completion interrupt. To stream data to the DSP, we use ping-pong double buffer within the DSP core's L2 SRAM and after the completion interrupt arrives, we swap the buffers and start filing the data to the other buffer.

IV. Proof of Concept

Because we are after a purely open-source approach, we can not make use of the TI proprietary DSPBIOS [10] RTOS and CCS cross compiler suite and IDE [11]. This means that we will need to build our own toolchains and runtime libraries. Fortunately, there are existing open-source project that can accomplish all of these tasks, the only problem is simply properly integrating them (and fixing the bugs in the process).

A. Development Setup

The development environment consists of a PC running Debian Linux unstable, the EVM board and two USRP N210s [12]. All of them are connected via a eight-port giga-bit ethernet switch. The USB-Serial converter embedded on the board is also connect to the PC for the serial terminal.

B. Components

1) ARM toolchain: The ARM toolchain is only needed to build the kernel and bootloader. That is, we need to be build `binutils` and `gcc`. Version 4.7.2 of gcc works fine for the purpose.

2) Linux Kernel: For this project we used the official TI kernel for the KeyStone II SoC. It is available from [13]. Remember to build the boot monitor `mon.bit` [14].

3) Linux Rootfs: For simplicity, we selected Debian unstable (sid) as the Linux Rootfs. We bootstrapped the system on the PC using the `debootstrap` [15] utility. This decision later turned out to be crucial when we started porting GNU Radio, because having a complete Linux distribution with very good package management eases the porting task.

4) Booting the System with NFS Root: Assuming the host Linux PC is on 172.16.1.1, and the board's IP is 172.16.1.2, the following `u-boot` transcript boots the system from NFS root (`/export/k2dsp`).

```
setenv serverip 172.16.1.1
setenv ipaddr 172.16.1.2
setenv bootargs  console=ttyS0,115200n8\
 rootwait=1 earlyprintk root=/dev/nfs\
 nfsroot=172.16.1.1:/export/k2dsp,v3,tcp\
 rw ip=dhcp
setenv _1 tftpboot 0c5f0000 mon.bin
setenv _2 mon_install 0x0c5f0000
setenv _3 tftpboot 87000000 evm.dtb
setenv _4 tftpboot 88000000 uImage
setenv _5 bootm 88000000 - 87000000
setenv _ run _1 _2 _3 _4 _5
```

The above port only need to be executed once on the board, and `saveenv` can save them into the flash. After that, starting the system is as simple as one command: `run _`.

It should be noted that by default `u-boot` reserve 512 MB of DDR for DSP use. Since we do not plan to use the DDR from the DSP, we set `etenv mem_reserve 128M` so that only 128MB is reserved. This makes more memory available for the ARMs and is convenient when compiling GNU Radio

5) Build GNU Radio: We used the latest git master branch of GNU Radio: https://github.com/gnuradio/gnuradio.git. To enable most of GNU Radio blocks, these following packages are needed:

```
libboost1.53-all-dev libcppunit-1.13-0 libcppunit-dev
libfftw3-dev libfftw3-3 python swig python-numpy
python-cheetah python-gtk2-dev python-lxml
python-wxgtk2.8 libgsl0-dev libzeroc-ice35-dev
```

6) DSP Toolchain: Except for a vague mentioning of c6x gcc at [16], there is not much information available on how to build the c6x backend of GCC [17].

After some trials and errors and digging through the `gcc` source code we discovered that the correct target is `c6x-none-elf`.

Only C language frontend can be built because `libc` is not yet available at this stadge and we do not plan to use C++ on DSP anyway.

```
../configure --target=c6x-none-elf \
 --enable-languages=c \
 --disable-shared --disable-libssp
```

7) DSP Runtime Libraries: To the best of our knowledge, a complete and free implementation of bare-mental C runtime library for C66x DSPs is not available. So we will have to port our own.

First we need to have a complete C runtime environment for the DSP. Devising the first program to run on the DSP is challenging because we do not have any way to access the DSP directly. Worse yet, we do not have any C runtime `crt0.o` available, so that the stack pointer is not setup correctly.

Therefore we carefully developed a C program, which does not require setting up the stack pointer (see Listing 1). The compiled program sends its status out through the serial port, which is guaranteed to be powered up and clocked.

Listing 1. First C Code Running on DSP

```
// c6x-none-elf-gcc -Os -fPIC -mdsbt -c test0.c -o test0.o
// c6x-none-elf-objcopy -O binary test0.o test0.bin
typedef unsigned int u32;
void _start() {
  volatile u32 *uart_data = (volatile u32 *)0x02530c00;
  *uart_data = '!';
  // won't return
  while (1) ;
}
```

Once Listing 1 is verified to work, we started adding support for a rudimentary C runtime using [18], as Listing 2.

Listing 2. General C Runtime

```
typedef unsigned int u32;
__attribute__((noinline)) void putchar(int c) {
    volatile u32 *uart_data = (volatile u32 *)0x02530c00;
    *uart_data = c;
}
__asm__ (
    ".section .text.boot,\"ax\",@progbits\n"
    ".global _start\n"
    "_start:\n"
    // setup the stack pointer (b15)
    "mvk .S2 0, b15\n"
    "nop 2\n"
    // 0x0C010000 = multicore shared memory + 64KB
    "mvkh .S2 0x0C010000, b15\n"
    "nop 2\n"
    "b .S1 (_init)\n"
    "nop 5\n"
    ".text\n"
);
__attribute__((noreturn)) void _init() {
    // clear .bss section
    extern char _edata[], _ebss[];
    extern int main();
    char *p = _edata;
    while (p < _ebss) *p++ = '\0';
    main();
}

__attribute__((noreturn)) int main(){
    putchar('?');
    // won't return
    while (1) ;
}
```

To build Listing 2, one will need a linker script (Listing 3) and makefile (Listing 4).

Listing 3. General C Linker Script

```
ENTRY(_start)
MEMORY {
    MCSM : ORIGIN = 0x0C000000, LENGTH = 6M
}
SECTIONS {
    .bin : {
        _text = .;
        *(.text.boot)
        *(.text) *(.text.startup)
        _etext = .;
        *(.data)
        _edata = .;
        *(.bss)
        _ebss = .;
    } > MCSM
}
```

Listing 4. The Makefile

```
CFLAGS := -Wall -Wextra -Werror -march=c674x \
          -fno-builtin \
          -O0 -ggdb
LDFLAGS := -Tlinker.map -nostdlib
all: test0.bin test1.bin
%.elf: %.o linker.map
        $(CC) $(LDFLAGS) -o "$@" $<
%.bin: %.elf
        $(OBJCOPY) -O binary "$<" "$@"
```

After getting the general C runtime environment working, we started porting the C library `libc` and the math library `libm`. Because there are a lot open-source libc implementations: musl[19], dietlibc[20], newlib[21], μClibc[22] and bionic for Android[23], we simply picked the one that we are most familiar with: newlib[21], we do not need a complete `libc` implementation as there is no OS running on the DSP.

For the math library, we have two choices, one is the fdlibm library open-sourced by Sun microsystem in 1993[24], the other is the one used in various BSD UNIX operating system, generally a fork of fdlibm[25]. We picked the latter, because it has been actively maintained in recent years. Having a complete IEEE-754 compliant `libm` is much more important than having a complete `libc`.

8) DSP Core Functions: For the DSP library, we use liquid-dsp [26], which is provides a standalone DSP library implemented in C with emphasis on embedded systems.

9) ARM control of the DSP: A less known fact of the IEEE POSIX pthread standard is that it does not actually require the thread to be running on the same processor architecture as the parent thread. We

devised a DSP controlling library that mimics the well-known pthread interface. For example, the user could use `pthread_create` to create a DSP thread on any of the available DSP cores. The entry address is actually a pointer to a relocatable ELF program file containing the DSP code. The argument to that thread is the address of the DSP L2 RAM where the input data resides. Another important API is `pthread_join`, which waits for the designated thread to terminate. This is achieved by waiting for the interprocessor interrupt that is sent from the DSP core (the `exit()` function for DSP code sends the interrupt to notify the completion of a program).

10) GNU Radio Integration: Following the model of [1], [6], [7] and [8], we created special DSP blocks in GNU Radio. The difference to existing approaches is that we use the foreign pthread interface discussed above and compile the DSP code on the fly. This enables assembling and disassembling of blocks that run on the DSP.

C. Results

We have completed all preceding steps. Since the latter part of project is still undergoing significant development and optimization, we will publish our performance benchmarks result at [7].

V. Conclusions

GNU Radio is a popular SDR development environment for education and research. This paper has described our initial work on porting GNU Radio to the Keystone II HMP-SoC. We argue for the need for fully open-source tools that support this development. Our long term vision is that of a unified scheduler that can dynamically partition waveform processing tasks and assign them to a suitable processor. This paper discussed the milestones on this mission and provides initial results. The code and supporting documents are available for free download from [27].

Acknowledgment

The authors would like to thank Texas Instruments for providing the of the cutting edge KeyStone II DSP+ARM SoC evaluation board. The authors also appreciate the support from the Wireless@VT lab for this work. Shenghou Ma is supported by the Configurable Computing Lab of Virginia Tech, and he would like to also thank his advisor Dr. Peter Athanas for his kind support on this work.

References

[1] A Standalone Package for Bringing Graphics Processor Acceleration to GNU Radio: GRGPU, http://gnuradio.org/redmine/attachments/download/257/06-plishker-grc-grgpu.pdf, 2011.

[2] C.H. Van Berkel, "Multi-core for mobile phones," in *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, 2009, pp. 1260–1265.

[3] EVMK2HX Evaluation Modules, http://www.advantech.com/Support/TI-EVM/EVMK2HX.aspx, 2013.

[4] Keystone 2 EVM Technical Reference Manual Version 1.0, http://wfcache.advantech.com/www/support/TI-EVM/download/XTCIEVMK2X_Technical_Reference_Manual_Rev1_0.pdf, 2013.

[5] TCI6638K2K: Multicore DSP+ARM KeyStone II System-on-Chip (SoC), http://www.ti.com/lit/ds/symlink/tci6638k2k.pdf, 2013.

[6] W. Plishker, G.F. Zaki, S.S. Bhattacharyya, C. Clancy, and J. Kuykendall, "Applying graphics processor acceleration in a software defined radio prototyping environment," in *22nd IEEE International Symposium on Rapid System Prototyping (RSP)*, 2011, pp. 67–73.

[7] Using Digital Signal Processors in GNU Radio, http://gnuradio.squarespace.com/storage/grcon13_presentations/grcon13_ford_snl_dsps.pdf, 2013.

[8] A. Love and P. Athanas, "Rapid modular assembly of Xilinx FPGA designs," in *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, 2013, pp. 1–1.

[9] TMS320C66x DSP Cache User Guide, http://www.ti.com/lit/ug/sprugy8/sprugy8.pdf, 2010.

[10] DSP/BIOS Real-Time Operating System (RTOS), http://www.ti.com/tool/dspbios.

[11] Code Composer Studio (CCStudio) Integrated Development Environment (IDE) v5, http://www.ti.com/tool/ccstudio.

[12] UHD - USRP2 and N2X0 Series Application Notes, http://files.ettus.com/uhd_docs/manual/html/usrp2.html.

[13] TI KeyStone SoC Linux Kernel, git://arago-project.org/git/projects/linux-keystone.git.

[14] MCSDK User Guide: Exploring the MCSDK, http://processors.wiki.ti.com/index.php/MCSDK_UG_Chapter_Exploring, 2013.

[15] Debootstrap, https://wiki.debian.org/Debootstrap.

[16] Linux-c6x Project, http://linux-c6x.org/.

[17] The GNU Compiler Collection, http://gcc.gnu.org/.

[18] The C6000 Embedded Application Binary Interface: Application Report, http://www.ti.com/lit/an/sprab89/sprab89.pdf, 2011.

[19] musl-libc: a new libc for linux, http://www.musl-libc.org/.

[20] diet libc: a libc optimized for small size, http://www.fefe.de/dietlibc.

[21] Newlib: a C library intended for use on embedded systems, http://sourceware.org/newlib/.

[22] μClibc: A C library for embedded Linux, http://www.uclibc.org/.

[23] bionic libc for Android, https://android.googlesource.com/platform/bionic.git.

[24] FDLIBM: Freely Distributable LIBM, http://www.netlib.org/fdlibm/.

[25] math library contributed by SunPro, http://svnweb.freebsd.org/base/head/lib/msun/.

[26] liquid-dsp: digital signal processing library for software-defined radios, http://liquidsdr.org.

[27] GNU Radio on DSP Project, https://github.com/GRDSP/.