Primate Architecture Reference

FAST Research Group

January 14, 2025

1 Primate

The long term intention of Primate is to generate Pareto-optimal solutions, including hardware and any necessary software, from sets of applications that spend most of their time in a common set of library calls/kernels. Primate will generate that hardware based on a set of specified metrics such as area, power, performance, flexibility, serviceability, and debugability. For example, Primate could be instructed to generate a solution that minimizes area and power, while maximizing flexibility and debugability, at the cost of performance, or maximize performance at the cost of area, power, flexibility, and debugability, or maximize debugabilty at the cost of all other metrics. Thus, Primate could generate a simple micro-coded RISC-V-based solution with debug counters, a dedicated circuit with no programmability (e.g., output of standard HLS), or a processor designed to provide cycle-by-cycle observability. The stretch goal of Primate is that all high performance application development is done on Primate first, even if the intended execution platform is a standard multi-core CPU.

A key motivation for Primate is the observation that much of high performance programming today is done by calling library functions, e.g., CuDNN and PyTorch, producing programs that run multiple orders of magnitude faster than standard sequential software. Those library functions, therefore, are key candidates for hardware acceleration. In addition, since performance improves by orders of magnitude, Amdahl's Law states that the vast majority of the computation is being done in the library calls.

This document describes the processor/C++ compiler generator subset of Primate which is what we are working on first. Our first targets are overlay processors implemented on FPGAs.

1.1 Quick Start Guide

Primate is shipped with a Docker image to aid in getting started quickly.

1. Clone the Primate parent repo: `git@github.com:FAST-Research-Group/primate.git`
2. Build the docker container `docker build -t primate-container .` [1] [2]
3. Run the docker container `docker run --name <meaningful_name> -d primate-container`
4. Attach to the docker container `docker exec -it <meaningful_name> bash`

Once you have a shell in the container, there are a few folders you should be made aware of. All primate code is contained in the `/primate/` folder. Under the primate folder there are three folders of interest: `primate-compiler` `primate-arch-gen` `primate-uarch`. `primate-arch-gen` and `primate-compiler` contains the source code for the archgen tool, and the compiler. You should be able to ignore these (unless you'd like to contribute). The `primate-uarch` directory contains everything required for generating new overlays. Looking in the `primate-uarch/apps` directory there are some included example projects. We suggest starting with the `example` project.

[1] Skip this step if the docker image is already built on your machine as it takes a while
[2] Requires around 32GB of DRAM. This builds LLVM.

A primate project is split into three main components: a `hw` folder, a `sw` folder, and a `build.sh` script. The `hw` folder contains all the `.scala` files for the blue functional units, as well as some configuration files we will detail later. The `sw` folder will contain the source code you wish to generate overlays for. Finally the `build.sh` script will run Archgen to create the new overlay, then compile the software and overlay resulting in a core that is ready to be simulated with verilator.

1.2 Running an Example

Primate comes with an example project which is a simple TCP parsing workload. This project can be found at `/primate/primate-uarch/apps/example`. To run the full primate flow, simply run `/primate/primate-uarch/apps/example/build.sh`. After this the folder should contain a file called `primate-pgm.bin` along with a `primate.cfg`. To run the verilog simulator navigate to `/primate/primate-uarch/chisel/Gorilla++/emulator` and run `make verilator`. This will run the simulation and generate a `Top.vcd` waveform in directory `/primate/primate-uarch/chisel/Gorilla++/test_run_dir/TopMain<RandomNumber>`

BOZO: Matthew: why does build.sh not generate verilog? Chisel generation flow should be separate from emulation (unless verilator backend provides great benefit)

If you run through this without error, then primate is set up correctly. Now we will give a brief overview of writing your own code, and adding your own hardware.

1.3 Writing New Software

Writing new primate code should be familiar to existing C++ developers. Developers should see Section 8 for more details about writing primate code, and Section 8.2.1 for a full list limitations. For now we will start with a simple example program which reads input from the streaming IO unit, increments a counter, and then outputs the new data to the IO unit.

All programs must include the primate common header:

```
#include "../../common/primate-hardware.hpp"
```

Then we need to define some structure for the data that we want to read. Lets assume for now its just a pair of ints:

```
struct DataFormat { int a; int b; };
```

Primate programs must start in the `primate-main` function which returns void, and takes no arguments

```
void primate-main()
```

Now we cans start writing our application logic which should be somewhat familiar to C++ developers. All together our program looks like:

```
#include "../../common/primate-hardware.hpp"

struct DataFormat { int a; int b; };

void primate-main() {
    // read from input stream
    struct DataFormat in = PRIMATE::input<struct DataFormat>();
    in.a++;
    // write to output stream
    Primate::output<struct DataFormat>(in);

    // clean up
    PRIMATE::input_done();
```

```
    PRIMATE::output_done();
}
```

To see a list of all the built in Blue Functional Units (BFUs) see Section 8.3. For interacting with custom BFUs see Section 8.2

Now we wish to compile the code to run on a primate core. First we must run Archgen and then the compiler. The steps to do so are listed below:

1. Make sure you have configured the hardware, and Archgen has been run (sw/primate.cfg exists, and hw/bfu_list.txt exist). If not configure the hardware (Section 4) and run Archgen now:

   ```
   primate-arch-gen/build/bin/clang -emit-llvm -S -O3 yourCode.cpp

   primate-arch-gen/build/bin/opt -enable-new-pm=0
   -load ${ARCH_GEN_DIR}/build/lib/LLVMPrimate.so
   -primate < yourCode.ll
   ```

2. Generate compiler configuration (in this example `bfu_list.txt` is an empty file):

   `primate-compiler/archgen2tablegen.py ../hw/bfu_list.txt ./primate.cfg`

3. copy the newly generated files into the compiler directory:

   ```
   cp ./primate-compiler-gen/IntrinsicsPrimate.td primate-compiler/
      llvm/include/llvm/IR/IntrinsicsPrimate.td
   cp ./primate-compiler-gen/PrimateInstrInfo.td primate-compiler/
      llvm/lib/Target/Primate/PrimateInstrInfo.td
   cp ./primate-compiler-gen/PrimateSchedPrimate.td primate-compiler/
      llvm/lib/Target/Primate/PrimateSchedPrimate.td
   cp ./primate-compiler-gen/PrimateSchedule.td primate-compiler/llvm
      /lib/Target/Primate/PrimateSchedule.td
   ```

4. rebuild the compiler (this may take a while)

   `ninja -C primate-compiler/build`

5. Use the compiler

   ```
   primate-compiler/build/bin/clang++ -I/lib/gcc/x86_64-linux-gnu/9/
      include/ -O3 --target=primate32-linux-gnu -march=pr32i -c <your
       code>.cpp -o primate_pgm.o
   ```

6. We do not have a loader (yet) so to simulate you must convert the ELF binary into a format we can load into the simulator:

   ```
   primate-compiler/build/bin/llvm-objdump -dr primate_pgm.o >
      primate_pgm_text
   primate-compiler/build/bin/llvm-objdump -t primate_pgm.o >
      primate_pgm_sym
   primate-compiler/bin2asm.py ./primate_pgm_text ./primate_pgm_sym
      ./primate_pgm.bin
   ```

1.4 Adding New Hardware

This section will detail the process of adding a new functional unit in scala/chisel. Primate code is written in scala/chisel, but if you'd like to use existing verilog you can write a wrapper around the verilog module in scala/chisel

TODO: section describing this process

Every interface to the units should be represented as chisel `Bundle`s to allow Primate to attach the ports required. Look at `./primate-uarch/apps/example/hw/match_table.scala` for an example

interface. All BFUs are required to follow the interface outlined in section 4.2. In summary the interface must be a bundle with the following signals:

```
in_valid    // is the input to the functional unit valid
in_tag      // the current thread ID making a request
in_opcode   // the incoming instruction opcode
in_imm      // the immediate field from the incoming instruction
in_bits     // the data for the request (from the register file)
in_ready    // if the input is ready to provide data (currently follows the valid signal)

out_valid   // is the output valid
out_tag     // the thread ID which made the currently completing request
out_flag    // optional program counter to jump to after completion (unsupported)
out_bits    // output data to the register file
out_ready   // if the ouput interface is ready to accept data
            // if this is false the functional unit should hold its current
            // output until true
```

Once you've written the new functional unit in the `hw` folder of your project, you need to tell primate the interface to use in the `bfu_list.txt`. Using the `matchTable` BFU from earlier as an example, we see the bundle Named `io` is the interface. We must create an entry in `bfu_list.txt` that looks like:

```
matchTable
{
    io
}
```

More generally you will create an entry

```
<BFU NAME>
{
    <interface 1 bundle name>
    <interface 2 bundle name>
    ...
    <interface n bundle name>
}
```

With newlines exactly as shown.

2 Build Flow

2.1 Overview

TODO: cleanup
A brief guide on how to run the primate workflow. This section should be simple and it should just work
TODO: make an example project?

The full primate build flow can be ran by executing the `primate-hardware`cn? target in the makefile in the primate-uarch directory.

in the directory where you wrote some primate code: run the flow with: make -f /primate/primate-uarch/scripts/Makefile

or alias that command so it will be easier to call:

the primate build flow can be broken down into 4 phases: running Archgen, running the compiler, building hardware, and setting up the simulation environment.

TODO: describe main organization of Makefile? (or is this too much?)
Each section will be described in more detail below.

2.2 ArchGen

Archgen is simply an LLVM pass which is run using `opt`. the output of Archgen is a configuration file called `primate.cfg`. This contains the parameters for the register file, thread state, and processor width. A detailed description of these fields in available in the following section.

2.2.1 ArchGen Parameters (primate.cfg)

ArchGen will generate the `primate.cfg` file detailing the parameters of the overlay. This section will go over each of the parameters in detail.

Below is a list of all the currently used parameters:

- `REG_WIDTH` determines the width of the register file in bits
- `REG_BLOCK_WIDTH` a space separated list of each register block width
- `NUM_REGBLOCKS` the length of the REG_BLOCK_WIDTH list
- `SRC_POS` the supported offsets into the register file. This is all the starting bit positions possible for a field in a register.
- `SRC_MODE` the supported sizes in the register file. This is all the field sizes possible for a field in a register. When combined with POS this should describe every possible field in the register file
- `MAX_FIELD_WIDTH` The largest SRC_MODE
- `NUM_SRC_POS` the length of the SRC_POS list
- `NUM_SRC_MODES` the length of the SRC_MODE list
- `DST_POS` same as SRC_POS but for writiing to the registers
- `DST_ENCODE` simply an index for all the DST_POS that are used by the hardware units
- `DST_EN_ENCODE` human readable version of the possible size, offset combinations that appear in the program. For example: 0 1 is position index 0, size index 1 where size index is from SRC_MODE
- `DST_EN` binary representation of the block enable required for a given offset, size pair. in the same order as DST_EN_ENCODE
- `NUM_DST_POS` length of the DST_POS list
- `NUM_WB_ENS` length of the DST_EN list
- `NUM_THREADS` number of threads the hardware supports
- `NUM_REGS` number of registers in the register file
- `NUM_ALUS` number of green functional units, a.k.a. RISC-V ALUs
- `NUM_BFUS` number of custom BFUs. Does not include the IO unit or the Load Store unit.
- `IP_WIDTH` width of the instruction pointer in bits
- `IMM_WIDTH` width of immediate fields in the instructions. measured in bits.

2.3 Compiler Gen

The compiler is based on Clang++ so running should be familiar to C++ developers. Before using the compiler you must generate a new compiler based on the output of Archgen. We provide a script to do this. Call it using `./primate-compiler/archgen2tablegen.py <bfu_list.txt> <primate.cfg>`. This will generate files in the directory `primate-compiler-gen` in the directory that the script was

invoked. Once this is completed those files should be copied into the compiler, and the compiler should be rebuilt to target your newly generated core. This can safely be copied from section 1.3. That section will also detail the required flags to invoke the compiler. (TODO: put those flags in the next section)

2.4 Compiling code for primate

TODO

2.5 Hardware Gen

TODO: what components are hardware gen dependent on?
bfu_list.txt (hand generated)
primate.cfg (archgen?)
...

2.5.1 Core template

Primate overlays start from a scala template at `hw/templates/primate.template`. Using the BFUs listed in `bfu_list.txt` and the parameters in `primate.cfg`, the `scripts/scm.py` script will instance any custom BFUs and wire the interfaces automatically. This script outputs valid chisel in the file `primtate.scala` that is ready for generation.

2.5.2 Chisel hardware generation

TODO: update paths/names once i figure out the flow for primate

The book "Digital Design with Chisel" is the best source on Chisel I've found.

To generate SystemVerilog, you must put the following code at the end of the Top.scala file:

```
object Top extends App {
  ChiselStage.emitSystemVerilogFile(new Top, Array("--target-dir", "
      generated"))
}
```

After running `sbt run` in the directory containing `build.sbt`, a SystemVerilog file will be generated at `generated/Top.sv`

2.6 Simulation/Testing

2.6.1 Tools/Methods/etc

necessary deps (TODO: workflow/ dep installation section?):

- java
- scala (chisel)
- verilator
- gtkwave
- python
- cocotb

ChiselSim seems unfinished and it requires us to write testbenches with the convoluted chisel syntax. A more robust (but currently more difficult in some ways. see 2.6.2) approach is to test and simulate using the generated SystemVerilog. This allows us to use well established workflows for verification. And hopefully will not require us to rewrite testbenches every time we upgrade to a newer version of Chisel.

The official simulator of Primate is Verilator. Testbenches can either be written in Verilator native C++, or in Python with the cocotb library. cocotb seems to be by far the easiest method for writing RTL testbenches. As a plus, it also supports many other simulation backends if we ever decide to switch from verilator. For these reasons, cocotb is the preferred method for writing primate tests.

2.6.2 Current issues

verilog generated by chisel loses some information useful for debugging. For example, Chisel Enums do not produce SystemVerilog Enums. Debugging things that use enums i.e.: evaluating the state of a FSM from simulation waveforms is very inconvenient.

Eventually Chisel will support generating enums. In the meantime, we may start using the Tywaves fork of chisel

2.6.3 testsuite/regression testing

TODO:
Write a suite of tests for base primate core.
Have all these tests run automatically

2.7 building for FPGA

Altera Quartus:

Xilinx Vivado:

3 ArchGen

3.1 Application Selection

3.1.1 ArchGen Parameters

moved to 2.2

3.1.2 BFU Generation

4 Hardware

4.1 Intruction Format

- Primate is a VLIW core. Each instruction sits in a specific slot which maps to a functional unit
- Branch unit instructions are in the last slot (high memory address or last slot)
- BFUs and GFUs can be merged. Starting from slot 0.
- The IO unit is always the last BFU Slot.
- The other BFUs are slotted in order they appear in bfu_list.txt
- if there is a GFU in a slot then we also must add 2 extracts and one insert. The insert is in the slot directly after the GFU. The extract is in the 2 slots directly before the GFU (Operand order is same as slot order).
- GFU instruction register indices must match the slot index of the extract they are reading from, inserts are agnostic to input register index

4.2 Interfaces

TODO: (how does the BFU interface work?) define/write after scalar registers

4.3 Template

TODO:
How does the chisel template work (put this in its own section)

4.4 ALU

ALU supports all RV32i R and U type ops also support a concatenate instruction (is this rv or primate spec?) rs1 and rd are the width of a register port (for now) for normal rv32i ops, the lower 32 bits of rd are the output of the alu, the upper [N:32] bits are just rs1 passed through. I guess this saves some mux bits? There is probably a more obvious way to write this, but im hoping to remove struct registers anyway so will probably just remove this logic.

4.5 Writing a new Blue Functional Unit

Primate blue functional units are simple.scala files. The module name and the file name MUST match.

5 Memory

The stated goal of primate is to run arbitrary programs. This section details the motivation for adding memory, the requirements for the memory, and the current implementation of that memory.

5.1 Motivation

Primate is able to scale its register file to support any data organization, and arbitrarily many registers. This may seem at first glance to remove the need for normal RAM, but there are still some key programming constructs that require a memory. Iterating over an array of data requires some dynamic indexing into an object which requires memory, or at least a dynamic index.

```
int sum = 0;

for(int i = 0; i < 500; i++)
    sum += data[i]
```

Memory also allows us to start emulating unsupported data. While primate aims to support all memory objects in native registers, an application change may render a structure as unsupported by the registers. This means we must emulate them in memory. This requires us to have the ability to operate on the object.

Finally memory allows for a level of indirection we cannot support without it. While object based memory will support this indirection, it will not support the emulation of unsupported data.

```
uint32_t idx = PRIMATE::Input<uint32_t>();

BFU_1(data[idx]);
```

In summary we need a memory that is able to emulate unsupported types, and able to be dynamically addressed.

5.2 Emulation

When a type is not supported in the register file, we must then emulate the type using memory. The requirements for this are quite simple. We need to be able to operate on the individual elements of the memory with shifts, and binary operations. We also need to guarantee a contiguous memory. That means we need the ability to

6 ISA

For a sull instruction listsing see

https://docs.google.com/spreadsheets/d/1zIkONqtpCCPnwrV5--QZSzdi75hy-1tzUBUt0h7arZU/edit

6.1 VLIW Packets

Primate's ISA is a Very Long Instruction Word (VLIW) ISA. Slot index is used to refer to the numbered sub instruction that makes up the VLIW instruction. Slot 0 is the instruction in the LSB of the VLIW word. slot N is the instruction in the MSBs of the VLIW word.

ALU Instructions have extracts and inserts. They will be refered to as ext and ins for the remainder of this section.

Consider a normal 3 address instruction: `add rs, rs1, rs2`. Our ISA requires ext operations for each source operand, and an ins operations for each destination operand. This means that GFU operations actually look like: EXT1, EXT2, ALU, INS. The slots from left to right will be n, n+1, n+2, n+3. *rs1 = EXT1*, *rs2 = EXT2*, *rd = INS*. This means that ALU instructions with only one register source will be encoded as *EXT, NOP, OP, INS*

Branch instructions are always in the highest slot index.

6.2 Field Specifiers

Extract and insert operations are used to specify a field to extract/insert from a structural register. While the encoding is specific to every generated architecture, there are some general rules that are good to know.

Field specifiers are split into *Mode* and *Position*. *Position* defines the bit that starts the field in the structured register. *Mode* defines the number of bits that make up the field. You can think of extracts as the following C snippet:

```
result = (source >> Position) & ((1 << Mode) - 1)
```

The field specifier is encoded as a concatenation of Position with Mode and kept in the immediate of the instruction. First a list of all possible Position and Mode values is computed. Then Mode and Position are converted into indecision into these lists. Then they are concatenated, with each field getting $clog2(len(list))$ bits. Pos lives in the high order bits and Mode lives in the low order bits of the immediate. They are encoded as the following C snippet:

```
imm = ((Pos_idx) << clog2(len(mode_list))) | (mode_idx)
```

6.3 Variable Register Indexing

Primate supports variable number of registers, and such supports variable number of bits in the register index field. Primate instructions will be rounded up to the nearest 8-bit byte. The fields remain in the same RISC-V ordering, but may contain padding between fields to ensure that decoding instructions doesn't require more bit swizzling than standard RISC-V. All field divisions will be padded to support the R-Type instruction encoding.

6.4 Merged Units

In order to save on register file ports we are able to merge units if there is an ALU and a BFU available. Merged units require inserts and extracts the same as ALUs. The extract and insert units have a field specifier for passing the entire register unmodified to support this. This is guaranteed to be generated, but is different every time so there is no specific field specifier for it.

7 Execution

7.1 Single Thread Execution

7.2 Inter-Processor communication

8 Software for Primate

Along with CPUs, the Primate framework also generates a collection of compilers. The generated compilers compile Primate C++ to Primate Machine Instructions. This section focuses on basics of writing Primate C++, along with subsections discussing performance considerations when writing Primate C++

8.1 By example

```
#include "primate-common.h"

typedef struct {
    int a, b, c;
} FMA_t;

#pragma primate blue FMA FMA 1 1
int fast_FMA(FMA_t work);

void p4_pipeline() {
    FMA_t networkWork;
    // get some data the IO unit.
    networkWork = PRIMATE::input<FMA_t>();
    // do out = a * (b + c);
    int res = fast_FMA(networkWork)
    // output the result to the IO unit.
    PRIMATE::output<int>(res);

    // clean up
    PRIMATE::output_done();
    PRIMATE::input_done();
}
```

Going line by line we start with the primate library include. This is required to use any primate intrinsics in the `PRIMATE` namespace. Next we see a definition of the `FMA_t` type which will contain the operands to do a fused multiply-add. It is simply 3 ints which will eventually be fed to the FMA unit which will do a*b+c. Next we see a definition of the `fast_FMA` function. This function corresponds to a hardware unit that we wish to instance in the overlay, so we will annotate it with `pragma primate blue <BFU Name> <instruction> <unit count> <latency>`. Finally, we have our application logic. This application reads input from the IO unit, which is a simple streaming data interface, performs the FMA, and outputs it to the IO unit.

8.2 Blue Functional Units

At the heart of the Primate Framework are Blue Functional Units. These units can be thought of as function calls in Primate C++. The software simply needs to call the BFU as a function, and the compiler will generate the required instructions to capture the live state, execute the BFU, and store the results in a register. The onus is on the software engineer to take any BFU side-effects into account (Stores to memory, early exits, registers changed besides the return value)

BFU functions require a forward declaration, and annotation. BFU functions are annotated with `pragma primate blue <BFU name> <instruction> <unit count> <latency>` to denote that the function is defined in hardware. `<BFU name>` is the name of the hardware unit which handles this

function (for if one BFU implements multiple functions). `<instruction>` defines the specific instruction this function maps to in the hardware BFU. You can think of this as the opcode for the BFU. The hardware writer will provide a mapping from instruction name to opcode, so the software programmer need not worry about that. `<unit count>` is the number of instances of this specific BFU the program will have. If there are conflicts the hardware generator will instance the largest number. `<clk>` is the average number of clock cycles to run the BFU. Once the forward declaration exists, the remaining application logic is written in vanilla C++.

8.2.1 Current Limitations

If you wish to change the code in the SW folder there are a few limitations to consider. Primate currently only supports inline functions, there isn't a set calling convention yet, and we suggest avoiding function calls in your code for now. This will change in the future.

Primate currently only support single module compiles, if you compile to .o files, and then link together the compiler will probably fail to generate code using the BFUs. The compile unit sent to the compiler needs to contain all BFU functions used in the entire program.

Primate does not support nested structures, so classes that contain classes, or structs that contain structs, will miscompile.

One thing that may appear but shouldn't is unstructured control flow. This is not supported and will probably never be supported. This is limited to `goto` for C++.

8.3 Primate Intrinsics

Along with BFU functions, Primate has some built in functions to interact with structural registers, and the Primate input unit. These can be found in the PRIMATE namespace.

`T PRIMATE::input<typename T>(int size=sizeof(T))` is a function to read data from the primate IO unit. After bytes are read they will be removed from the buffer. You can override the size to change the number of bytes to read from the buffer.

`void output<T>(T out, int size=sizeof(T))` is a function to output data to the primate IO unit. Bytes will be packed before sending to the interface by the hardware. Users can override the size to change the number of bytes written to the buffer.

9 Bug Reporting

Primate is still a work in progress, and so you may encounter bugs! This section will detail how to report those bugs. These bug reports can be made as issues on the Primate repository on github: https://github.com/FAST-Research-Group/primate

9.1 Compiler Bugs

Primate's compiler is built using clang, and by default uses a debug build of clang++. This means you can simply add the flags `-mllvm -debug` to your clang++ command and this will spit out all the relevant logs. Send us the output generated, and that will contain everything we need from your end.

9.2 Hardware Bugs

Primate simulations will generate VCD files that we can use to debug. In addition to the VCD we ask that you send us the same output from the compiler as in section 9.1.

10 Glossary

Term	Definition
Blue Functional Unit (BFU)	Domain specific functional unit. These are the user provided custom logic functional units that allow Primate to accelerate workloads.
Green Functional Unit (GFU)	Standard RISC-V ALU Functional units. These are automatically placed in the design and allow Primate generated Overlays to execute standard CPU code (glue logic, exception paths, etc.)

11 Papers on Overlays

OPU: An FPGA-Based Overlay Processor for Convolutional Neural Networks

```
root@91ea060aead3:/primate/test# /primate/primate-compiler/build/bin/llvm-objdump -S --arch-name=primate32 ./build/primate_pgm.o
```

Figure 1: This command outputs the assembly separated by instruction width of a specific binary file

```
root@91ea060aead3:/primate# cd ../
root@91ea060aead3:/# ls
bin   dev  home  lib32  libx32  mnt  primate  root  sbin  sys  usr
boot  etc  lib   lib64  media   opt  proc     run   srv   tmp  var
root@91ea060aead3:/# cd ../
root@91ea060aead3:/# ls
bin   dev  home  lib32  libx32  mnt  primate  root  sbin  sys  usr
boot  etc  lib   lib64  media   opt  proc     run   srv   tmp  var
root@91ea060aead3:/# cd primate
root@91ea060aead3:/primate# ls
Dockerfile  create_image.sh  deps              primate-compiler
README.md   dep              primate-arch-gen  primate-uarch
root@91ea060aead3:/primate# cd ../hw
bash: cd: ../hw: No such file or directory
root@91ea060aead3:/primate# cd ../
root@91ea060aead3:/# cd primate
root@91ea060aead3:/primate# ls
Dockerfile  create_image.sh  deps              primate-compiler
README.md   dep              primate-arch-gen  primate-uarch
root@91ea060aead3:/primate# mkdir test
root@91ea060aead3:/primate# cd test
root@91ea060aead3:/primate/test# vim bfu_list.txt
root@91ea060aead3:/primate/test# cd ../
root@91ea060aead3:/primate# cd test
root@91ea060aead3:/primate/test# ls
bfu_list.txt
root@91ea060aead3:/primate/test# pwd
/primate/test
root@91ea060aead3:/primate/test# export PRIMATE_ROOT=^C
root@91ea060aead3:/primate/test# export PRIMATE_ROOT=/primate/test
root@91ea060aead3:/primate/test# make -f ../primate-uarch/Makefile primate-sim
```

Figure 2: Setting up a new project to get binary

12 New uArch