NC STATE UNIVERSITY
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

ECE 463/521: FALL 2016: PROJECT 3: Dynamic Instruction Scheduling

Due: Tuesday November 29th, 11:59 PM

1. Ground rules

- This is an *individual* project.
- Collaboration by sharing code is strictly prohibited and is considered cheating.
- A TA will scan code from current *and past* semesters through automated tools available to us that can detect cheating; code that is flagged by these tools will be dealt with severely and will receive appropriate action in accordance with university policy.
- Use the tag Project3 in the Discussion Forum for questions (or write to TA/instructor).
- Don't submit any *code* in the Discussion Forum (check with TA or instructor if in doubt)
- You *must* use C/C++ programming languages. Exceptions must be approved by TA or instructor.
- Compatibility with EOS Linux environment is *required*.

2. Project Summary

In this project, you will construct a simulator for an out-of-order superscalar processor that fetches and issues *N* instructions per cycle. Only the dynamic scheduling mechanism will be modeled in detail, *i.e.*, perfect caches and perfect branch prediction are assumed. The imperfect instruction cache with/without next-line prefetching is optional for students who want to gain extra credits. The rest of this document describes precisely what you will implement (Specification), how you will test your simulator (Validation runs), how will you communicate your findings (Report) and what all you will submit (Report, source code etc.).

3. Specification

3.1. Microarchitecture to be modeled

3.1.1. Parameters

1. Number of architectural registers: The number of architectural registers specified in the ISA is 67 (r0-r66).[1] The number of architectural registers determines the number of entries in the Rename Map Table (RMT) and Architectural Register File (ARF).
2. WIDTH: This is the superscalar width of all pipeline stages, in terms of the maximum number of instructions in each pipeline stage. The one exception is Writeback: the number of instructions that may *complete* execution in a given cycle is *not limited* to WIDTH (this is explained below).
3. IQ_SIZE: This is the number of entries in the Issue Queue (IQ).
4. ROB_SIZE: This is the number of entries in the ReOrder Buffer (ROB).
5. CACHE_SIZE: This is the instruction cache size in bytes.
6. P: This is the prefetching flag indicating whether the next-line prefetching is enabled.

Note that 5 and 6 are 0 by default for all ECE463/521 students. See Section 10 for grading details.

[1] The ISA is MIPS-like: 32 integer registers, 32 floating-point registers, the HI and LO registers (for results of integer multiplication/divide), and the FCC register (floating-point condition code register).

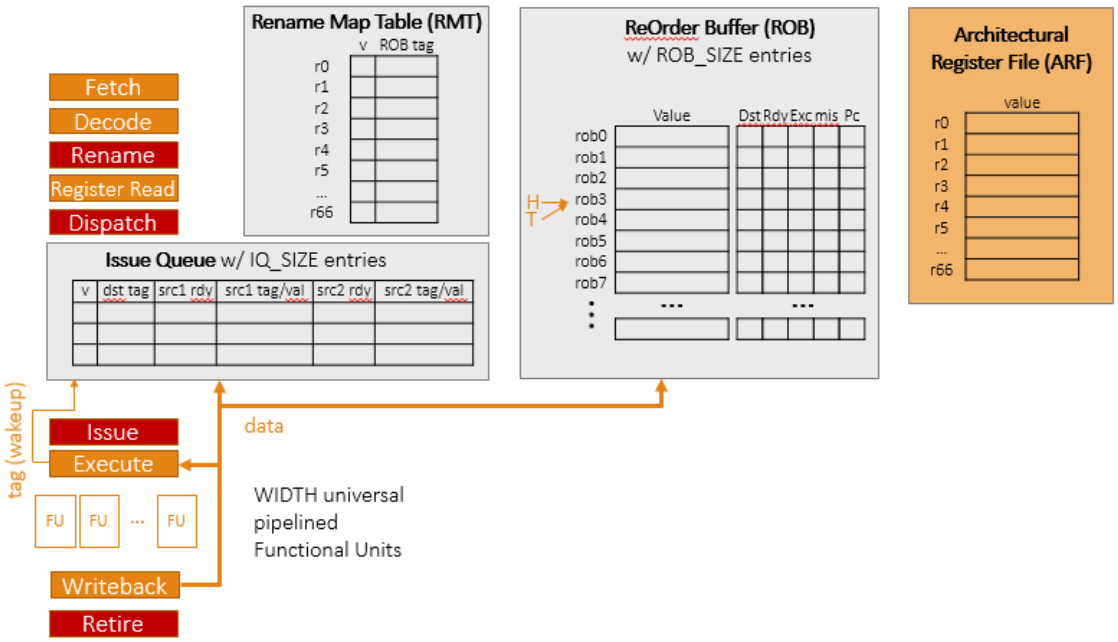

Figure 1. Overview of microarchitecture to be modeled, including the terminology and parameters used throughout this specification.

3.1.2. Functional Units

There are WIDTH number of *universal*, *pipelined* function units (FUs). Each FU can execute *any* type of instruction (hence the term "universal"). The operation type of an instruction indicates its execution latency: Type 0 has a latency of 1 cycle, Type 1 has a latency of 2 cycles, and Type 2 has a latency of 5 cycles. Each FU is fully pipelined. Therefore, a new instruction can begin execution on each FU every cycle.

3.1.3. Pipeline Registers

The pipeline stages shown in Figure 1 are separated by pipeline registers. In general, this specification names a pipeline register based on the stage that it feeds into. For example, the pipeline register between Fetch and Decode is called DE because it feeds into Decode. A *bundle* is the set of instructions in a pipeline register. For example, if DE is not empty, it contains a *decode bundle*. Table 1 lists the names of the pipeline registers used in this spec. It also provides a description of each pipeline register and its size (max # instructions).

Table 1. Names, descriptions, and sizes of all of the pipeline registers.

Pipeline Register	Description	Size (max # instructions)
DE	pipeline register between the Fetch and Decode stages	WIDTH
RN	pipeline register between the Decode and Rename stages	WIDTH
RR	pipeline register between the Rename and Register Read stages	WIDTH
DI	pipeline register between the Register Read and Dispatch stages	WIDTH
IQ	Queue between the Dispatch and Issue stages	IQ_SIZE

execute_list	execute_list represents the pipeline register between the Issue and Execute stages, as well as all sub-pipeline stages within each function unit.	WIDTH*5 WIDTH FUs each with a max. latency of 5 cycles. Hence, there can be as many as WIDTH*5 instructions in-flight within the Execute stage.
WB	pipeline register between the Execute and Writeback stages. *To maintain a non-blocking Execute stage, there is no constraint on the number of instructions that may complete in a cycle.*	WIDTH*5 This is a conservative upper bound. If each function unit's 5-stage pipeline is full with a 1-cycle (youngest), 2-cycle, 3-cycle, 4-cycle, and 5-cycle (oldest) operation, then all 5 instructions will complete in the same cycle. Multiply that by the number of such function units (WIDTH).
ROB	Queue between the Writeback and Retire stages	ROB_SIZE

A note about register *values*:
For the purpose of determining the number of cycles it takes for the microarchitecture to run a program, the simulator does *not* need to use and produce actual register values. This is why the initial Architectural Register File (ARF) values are not provided and the instruction opcodes are omitted from the trace. All that the simulator needs, to determine the number of cycles, is the microarchitecture configuration, execution latencies of instructions (by operation type), and register specifiers of instructions (to determine true, anti, and output dependencies).

3.1.4. Instruction Cache (Optional for all ECE463/521 students)

There is an instruction cache in the simulator that provides instructions for the fetch stage. Instruction PC is used to access the instruction cache. Each instruction is 32 bits wide. The instruction cache block size is 64 bytes and 4-way set associative with LRU replacement policy. The cache size is an input parameter. On a cache hit, the fetch stage takes 1 cycle. Otherwise, the fetch stage will be blocked/stalled for 10 cycles (i.e. new instructions cannot be fetched for the next 10 cycles). Note that the rest of the pipeline is not blocked. So if there are instructions waiting in decode, rename, register read, dispatch, issue, execution, writeback and retire stages, they will continue normal execution.

3.1.5. Next-line Prefetching (Optional for all ECE463/521 students)

The instruction cache is equipped with a next-line prefetcher. On an instruction cache miss to block address B, the next-line prefetcher will bring block B+1 into the cache if it is not already in the cache (if it is in the cache, nothing happens). If the processor uses a prefetched block P (e.g. prefetch hit), the prefetcher will bring block P+1 into the cache if it is not already in the cache (if it is in the cache, nothing happens). Note that in this case the fetch stage is not stalled but the prefetched block should not be accessible until 10 cycles memory access latency. For example, at cycle 200, a prefetch request is issued for block P. If the processor requests, P at cycle 205, then the fetch stage has to stall until cycle 210 which is when P becomes available. To implement this feature, you need to store with each block a timer indicating when they will become ready for access. To differentiate prefetched blocks from those that brought in by demand misses, mark prefetched blocks with a flag in the cache. When the processor uses a prefetched block, or if a prefetched block gets replaced, the prefetch flag is cleared.

3.2. Guide to Implementing your Simulator

This section provides a guide to implementing your simulator. Call each pipeline stage in reverse order in your main simulator loop, as follows. The comments indicate tasks to be performed.

```
do {
  retire();           // Retire up to WIDTH consecutive "ready" instructions from the head of
                      // the ROB.

  writeback();        // Process the writeback bundle in WB: For each instruction in WB, mark
                      // the instruction as "ready" in its entry in the ROB.

  execute();          // From the execute_list, check for instructions that are finishing
                      // execution this cycle, and:
                      // 1) Remove the instruction from the execute_list.
                      // 2) Add the instruction to WB.
                      // 3) Wakeup dependent instructions (set their source operand ready
                      // flags) in the IQ, DI (dispatch bundle), and RR (the register-read
                      // bundle).

  issue();            // Issue up to WIDTH oldest instructions from the IQ. (One approach to
                      // implement oldest-first issuing is to make multiple passes through
                      // the IQ, each time finding the next oldest ready instruction and then
                      // issuing it. One way to annotate the age of an instruction is to
                      // assign an incrementing sequence number to each instruction as it is
                      // fetched from the trace file.)
                      // To issue an instruction:
                      // 1)      Remove the instruction from the IQ.
                      // 2)      Add the instruction to the execute_list. Set a timer for
                      //         the instruction in the execute_list that will allow you to
                      //         model its execution latency.

  dispatch();         // If DI contains a dispatch bundle:
                      // If the number of free IQ entries is less than the size of the
                      // dispatch bundle in DI, then do nothing. If the number of free IQ
                      // entries is greater than or equal to the size of the dispatch bundle
                      // in DI, then dispatch all instructions from DI to the IQ.

  regRead();          // If RR contains a register-read bundle:
                      // If DI is not empty (cannot accept a new dispatch bundle), then do
                      // nothing. If DI is empty (can accept a new dispatch bundle), then
                      // process (see below) the register-read bundle and advance it from RR
                      // to DI.
                      //
                      // How to process the register-read bundle:
                      // Since values are not explicitly modeled, the sole purpose of the
                      // Register Read stage is to ascertain the readiness of the renamed
                      // source operands. Apply your learning from the class lectures/notes
                      // on this topic.
                      //
                      // Also take care that producers in their last cycle of execution
                      // wakeup dependent operands not just in the IQ, but also in two other
                      // stages including RegRead()(this is required to avoid deadlock). See
                      // Execute() description above.

  rename();           // If RN contains a rename bundle:
                      // If either RR is not empty (cannot accept a new register-read bundle)
                      // or the ROB does not have enough free entries to accept the entire
                      // rename bundle, then do nothing.
                      // If RR is empty (can accept a new register-read bundle) and the ROB
                      // has enough free entries to accept the entire rename bundle, then
                      // process (see below) the rename bundle and advance it from RN to RR.
                      //
                      // How to process the rename bundle:
                      // Apply your learning from the class lectures/notes on the steps for
                      // renaming:
                      // (1) Allocate an entry in the ROB for the instruction,
                      // (2) Rename its source registers, and
                      // (3) Rename its destination register (if it has one).
```

```
                        // Note that the rename bundle must be renamed in program order.
                        // Fortunately, the instructions in the rename bundle are in program
                        // order).

    decode();           // If DE contains a decode bundle:
                        // If RN is not empty (cannot accept a new rename bundle), then do
                        // nothing. If RN is empty (can accept a new rename bundle), then
                        // advance the decode bundle from DE to RN.

    fetch();            // Do nothing if instruction cache is perfect (CACHE_SIZE=0) and either
                        // (1) there are no more instructions in the trace file or
                        // (2) DE is not empty (cannot accept a new decode bundle).
                        //
                        // If there are more instructions in the trace file and if DE is empty
                        // (can accept a new decode bundle), then fetch up to WIDTH
                        // instructions from the trace file into DE. Fewer than WIDTH
                        // instructions will be fetched only if the trace file has fewer than
                        // WIDTH instructions left.
                        //
                        // If instruction cache is imperfect or next-line prefetcher is
                        // enabled, in addition to the above operations, adjust the timer to
                        // model fetch latency when necessary.

} while (advance_cycle());
                        // advance_cycle() performs several functions.
                        // (1) It advances the simulator cycle.
                        // (2) When it becomes known that the pipeline is empty AND the trace
                        // is depleted, the function returns "false" to terminate the loop.
```

4. Simulator Input

The simulator reads a trace file in the following format:

```
<PC> <operation type> <dest reg #> <src1 reg #> <src2 reg #>
<PC> <operation type> <dest reg #> <src1 reg #> <src2 reg #>
...
...
```

Where:

- <PC> is the program counter of the instruction (in hex).
- <operation type> is either "0", "1", or "2".
- <dest reg #> is the destination register of the instruction. If it is **-1**, then the instruction does not have a destination register (for example, a conditional branch instruction). Else, it is between 0 and 66.
- <src1 reg #> is the first source register of the instruction. If it is **-1**, then the instruction does not have a first source register. Else, it is between 0 and 66.
- <src2 reg #> is the second source register of the instruction. If it is **-1**, then the instruction does not have a second source register. Else, it is between 0 and 66.

For example:

```
ab120024  0   1  2  3
ab120028  1   4  1  3
ab12002c  2  -1  4  7
```

Means:

"operation type 0" R1, R2, R3

"operation type 1" R4, R1, R3

"operation type 2" -, R4, R7 *// no destination register!*

5. Simulator Output

The simulator outputs the following measurements after completion of the run:

1. Total number of instructions in the trace.
2. Total number of cycles to finish the program.
3. Average number of instructions retired per cycle (IPC).
4. Total number of instruction cache hits in the trace.
5. Total number of prefetch hits in the trace.

Note that you don't need to print out 4 and 5 by default when <CACHE_SIZE> and <P> are both 0.

The simulator also outputs the timing information for every instruction in the trace, in a format that is used as input to the scope tool. The scope tool's input format is described in a later section.

6. Helping you debug and validate

We provide a *scope* tool written by Dr. Eric Rotenberg that allows you to display pipeline timing diagrams similar to the timing diagrams used in class.

Usage: **`scope <input-file> <output-file>`**

The tool has quite a bit of error checking to make sure you comply with formatting and usage, however, beware it is not error-proof. The input to *scope* must be a text file that encodes the timing of each instruction in the program. Your simulator must dump this timing information and generate the input-file (needed anyway to match validation output). There should be one line for each instruction in the program, and instructions must be dumped in program order. The format of each line is as follows. Note: you must substitute an integer everywhere there is a <> pair.

```
<seq_no> fu{<op_type>} src{<src1>,<src2>} dst{<dst>} FE{<begin-cycle>,<duration>} DE{…} RN{…}
RR{…} DI{…} IS{…} EX{…} WB{…} RT{…}
```

<seq_no> is the line number in trace (*i.e.*, the dynamic instruction count), starting at 0. Substitute 0, 1, or 2 for the <op_type>. <src1>, <src2>, and <dst> are register numbers (include –1 if that is the case). For each of the pipeline stages, indicate the first cycle that the instruction was in that pipeline stage followed by the number of cycles the instruction was in that pipeline stage. The tool automatically does some consistency checks and exits if there is a problem, *e.g.*, begin cycle of DE should equal begin cycle of FE plus duration of FE. The <output-file> generated by *scope* contains the timing diagram.

7. Compiling, running and further validating the simulator

7.1. Compiling the simulator source

Part of what you need to hand in is source code. The TA will compile and run your simulator. As such, you must meet the following strict requirements. Failure to meet these requirements will result in point deductions (see Section 10).

1. You must be able to compile and run your simulator on Eos Linux machines. This is required so that the TA can compile and run your simulator. If you are logging into an Eos machine remotely and do not know whether or not it is Linux (as opposed to SunOS), use the *uname* command to determine the operating system.
2. Along with your source code, you must provide a Makefile that automatically compiles the simulator. This Makefile must create a simulator named *sim_bp*. The TA should be able to type only *make* and the simulator should compile successfully. The TA should be able to type only *make clean* to automatically remove object (.o) files and the simulator executable. An example Makefile will be posted on the web page; you can copy and modify it for your needs.

7.2. Running the simulator binary

Your simulator must accept command-line arguments as follows:

```
sim_ds <ROB_SIZE> <IQ_SIZE> <WIDTH> <CACHE_SIZE> <P> <tracefile>
```

7.3. Validating the simulator output

Sample simulation outputs will be provided on the website.
Each validation run includes:

1. Timing information for every instruction. The format is described in Section 6.
2. The microarchitecture configuration.
3. All measurements as described in Section 5.

You must run your simulator and debug it until it matches these provided *validation outputs*. Your simulator must print outputs to the console (i.e., to the screen). Your output must match both numerically and in terms of formatting because the TA will literally "diff" your output with the correct output. Therefore, redirect the console output of your simulator to a temporary file. This can be achieved by placing

```
> <your_output_file>
```

after the simulator command. You can test whether or not your outputs match the expected output, by running this unix command:

```
diff –iw <your_output_file> <posted_output_file>
```

The –iw flag tells diff to ignore case (uppercase vs. lowercase) and whitespace. Therefore, just make sure there is some whitespace (such as a tab) where you see whitespace in the validation output. If the above command returns without printing anything to the screen, your validation was successful.

8. Experiments and Report

8.1. Effects of IQ_SIZE with a large ROB

Plot1 Keep ROB_SIZE fixed at 512 entries so that it is not a resource bottleneck. For each benchmark, plot IPC on the y-axis and IQ_SIZE on the x-axis; use IQ_SIZEs of 8, 16, 32, 64, 128, and 256. Plot 4 different curves on each graph: one curve for each of WIDTH = 1, 2, 4, and 8. You may name the plots 1a, 1b, 1c etc. per benchmark.

Analysis Using the data in the plot above, for each WIDTH (1, 2, 4, and 8), find the minimum IQ_SIZE that still achieves an IPC no worse than 5% less than the IPC with the largest IQ_SIZE (256). This exercise should give four optimized IQ_SIZEs per benchmark, one optimized for each of WIDTH = 1, 2, 4, and 8. Tabulate the results of this exercise as follows:

	Optimized IQ_SIZE per WIDTH Minimum IQ_SIZE that still achieves within 5% of the IPC of the largest IQ_SIZE		
	Benchmark 1	Benchmark 2	…
WIDTH = 1			
WIDTH = 2			
WIDTH = 4			
WIDTH = 8			

Discussion

- The goal of a superscalar processor is to achieve an IPC close to the WIDTH of the machine. Given this goal, what is the relationship between WIDTH and IQ_SIZE? Explain.
- Do some benchmarks show higher or lower IPC than other benchmarks for the same microarchitecture configuration? Why might this be the case?

8.2. Effect of ROB_SIZE

Plot2 For each benchmark, make a plot with IPC on the y-axis and ROB_SIZE on the x-axis; use ROB_SIZE = 32, 64, 128, 256, and 512. Plot 4 different curves on the graph: one curve for each of WIDTH = 1, 2, 4, and 8. For each WIDTH, use the optimized IQ_SIZE for that WIDTH as obtained from the table in Section 8.1.

8.3. Effect of Instruction Cache (Optional for all ECE463/521 students)

Plot3 For each benchmark, make a plot with IPC on the y-axis and CACHE_SIZE on the x-axis; Use ROB_SIZE = 32, IQ_SIZE=64 and WIDTH=4. Plot 4 different curves on the graph: one curve for each of CACHE_SIZE = 4, 8, 16, and 32KB. How does the instruction cache help/impede with ILP?

8.4. Effect of Prefetching (Optional for all ECE463/521 students)

Plot4 Repeat the cache size variation experiment in Section 8.3 above with prefetching (set <P> to 1). Does it help? Can you achieve similar results with a smaller cache?

9. What to Submit via Wolfware

You must hand in a single zip file called **proj3.zip** that is no more than 1MB in size. Please respect the size limit on behalf of fellow students as the Wolfware submission space is limited. Notify the TA beforehand if you have special space requirements. However, a zip file of 1MB should be sufficient. Below is an example showing how to create proj3.zip from an Eos Linux machine. Suppose you have a bunch of **source code files** (*.cc, *.h), the **Makefile**, and your project report (**report.pdf**). Run "zip proj3 *.cc *.h Makefile report.pdf"

proj3.zip must only contain the following (any deviation may result in point deductions):

- Project report: This must be a single PDF document named report.pdf (or a single MS Word document called report.doc). The report must include a cover page with the project title, the Honor Pledge, and your full name as electronic signature of the Honor Pledge. A sample cover page will be posted on the project webpage. See Section 8 for the content required in the report.
- Source code: You must include the commented source code for the simulator program itself. You may use any number of .cc/.h files, .c/.h files, etc.
- Makefile: A sample Makefile will be posted on the project webpage. See Section 7.1 for what is expected of the makefile.

Note: Zip (no tar) only the *files* listed above, not the directory containing these files.

10. Grading

The following is the high level break-up of the points awarded.

0	If you do not hand in anything by the due date
+40	Your simulator does not compile, run or work, but you hand in significant+commented code
+40	For correct validation (8 validations + 2 mystery tests with 4 points each)
+20	For the report (plots, analysis, discussion), provided your simulator validates correctly first.
+20	For instruction cache (1 validation + 1 mystery tests + plots & analysis). The validation, including working code, is 10 points.
+20	For instruction cache prefetch (1 validation + 1 mystery tests + plots & analysis). The validation, including working code, is 10 points.

11. Deductions

- -10 point for each hour late, according to Wolfware timestamp. TIP: Submit whatever you have completed by the deadline just to be safe. If, after the deadline, you want to continue working on the project to get more features working and/or finish experiments, you may submit a second version of your project late. The TA will grade both the on-time and late versions (with late penalty applied to the late version), and use the one scoring more as the official submission. That said, please try and complete everything by the deadline.
- Up to -20 points for not complying with specific procedures. Follow all procedures very carefully to avoid penalties.
- Cheating: Source code that is flagged by tools available to us will be dealt with according to University Policy. This includes a 0 for the project and referral to the Office of Student Conduct.

12. Other Recommendations

12.1. Keep your simulator parametric

Always assume that parameters are changeable. As an example, you should not assume a specific WIDTH, ROB_SIZE or IQ_SIZE etc. Such parameters should be parsed from the input of the simulator and then used to instantiate an instance of the generic class.

12.2. Keep backups

It is good practice to frequently make backups of all your project files, including source code, your report, etc. You can backup files to another hard drive (Eos account vs. home PC vs. laptop vs. USB memory stick etc.). Just keep consistent copies in multiple places.

12.3. Make your simulator fast

Correctness of your simulator is of paramount importance. That said, making your simulator *efficient* is also important for a couple of reasons. First, the TA needs to test every student's simulator. Therefore, we are placing the constraint that your simulator must finish a single run in 2 minutes or less. If your simulator takes longer than 2 minutes to finish a single run, please see the TA. Second, you will be running many experiments: many pipeline configurations and multiple traces. Therefore, you will benefit from implementing a simulator that is reasonably fast. One simple thing you can do to make your simulator run faster is to compile it with a high optimization level. The example Makefile posted on the web page includes the –O3 optimization flag. Note that when you are debugging your simulator in a debugger (such as gdb), it is recommended that you compile without –O3 and compile with –g. The –g flag tells the compiler to include symbols (variable names, etc.) in the compiled binary. When you are done with debugging, recompile with –O3 and without –g.

12.4. Use the VCL

In addition to using the grendel cluster, and other Eos linux machines, NCSU's Virtual Computing Lab. (VCL) allows you to reserve virtual linux machines. Go to http://vcl.ncsu.edu/

1. select "Reservation > New Reservation",
2. you'll see a popup asking for environment and duration,
3. for the environment, select "Linux Lab Machine (Realm RHEnterprise 6)"
4. you may choose to reserve a shell for up to 4 hours beginning now or later (with the possibility to extend reservation before it expires),
5. click "Create Reservation" button,
6. wait until the screen updates with a reservation and click "Connect!",

7. You will be provided an internet address that you can then ssh into using putty or other ssh clients (i.e. the same way that you remotely and securely login to other linux machines).