Lab 6 Report

Huy Lai — 132000359
Kyle Crusius — 428003578
Arjun Kurkal — 631004597
Vishnu Naidu — 630004567

Texas A&M University

13 December 2022

An Aggie does not lie, cheat or steal.
Nor does an Aggie tolerate those who do.

Our Eternal thanks to **Kevin Weston** for helping us decipher this entire CPU.

Fetch

The Fetch stage reads the instructions from the instructions memory and places the arguments into `iCode`, `iFun`, `rA`, `rB`, and `valC`.

The `FetchRegEnable` sub-circuit will allow the Fetch stage to load the value `iCode`, `iFun`, `rA`, and `rB`. The `FetchDataEnable` sub-circuit will allow the Fetch state to load the value `valC`. Both of these circuits will also handle the varing lengths of instructions and load the appropriate bytes into the output.

The `FetchValP` sub-circuit will allow the Fetch state to load the value `valP`

Additionally, when the Fetch stage is done loading the instruction and its required values, a `complete` pulse is sent. Since the Fetch stage needs to load between one and 10 bytes, the complete flag, using `valP`, will dynamically wait until all the values are loaded.

Contributors

Huy Lai
Arjun Kurkal
Kevin Weston

Screenshots

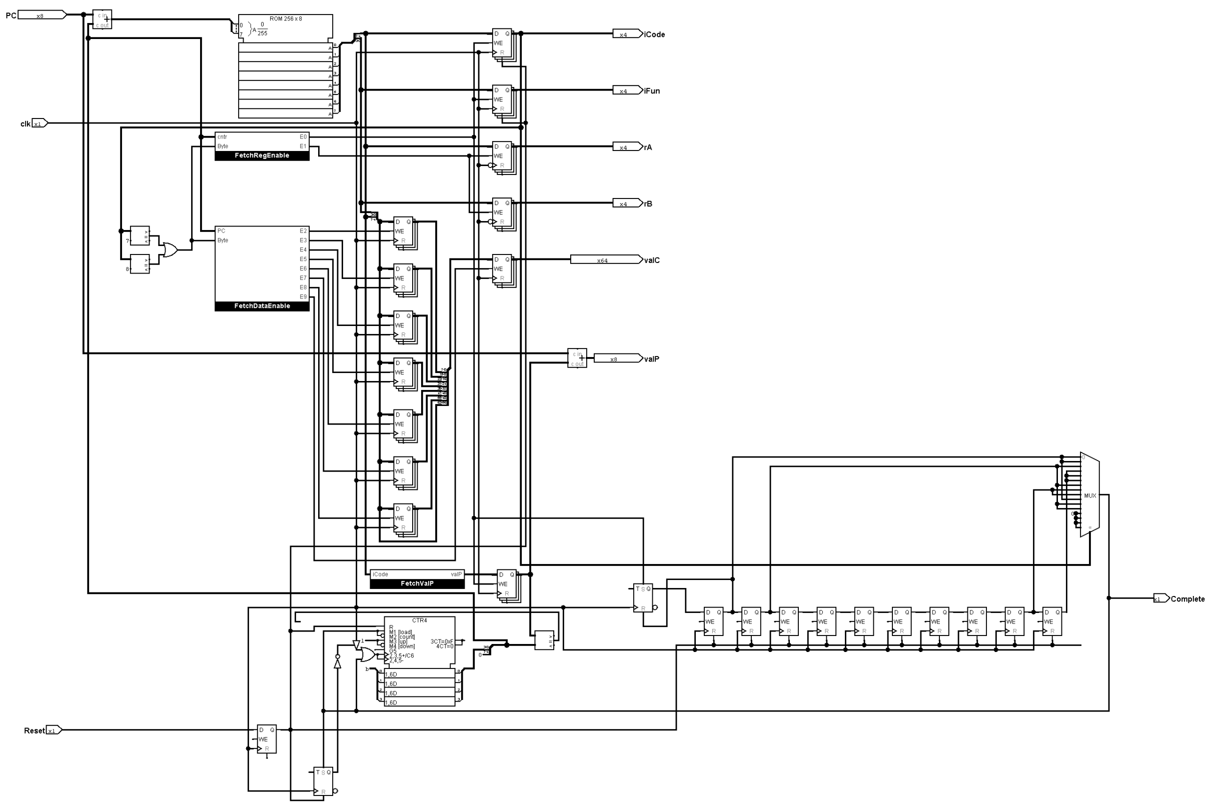

Figure 1: Fetch

Decode

The Decode block provides access to the register file, as well as interpreting instructions from Fetch to regulate read and access. The Decode block accepts `iCode`, `rA`, `rB`, `valM`, `valE`, `dstM`, and `dstE` as input (as well as a clock signal). It returns `valA` and `valB`.

Decode is comprised of three primary subsections: the register file, `srcA`, and `srcB`.

The register file provides output to `valA` and `valB` based on the registers it is instructed to read. Using `iCode`, `srcA` and `srcB` determine if a register needs to be read, and passes along the address from `rA` or `rB` respectively if so.

The register file can also be modified using `dstM`, `dstE`, `valM`, and `valE`. If the `WriteEnable` is enabled, the register file will update the corresponding destination with its appropriate value. The `MEDecoder` helps determine which of E or M should be written to, since only one can happen at a time.

Contributors

Kyle Crusius

Screenshots

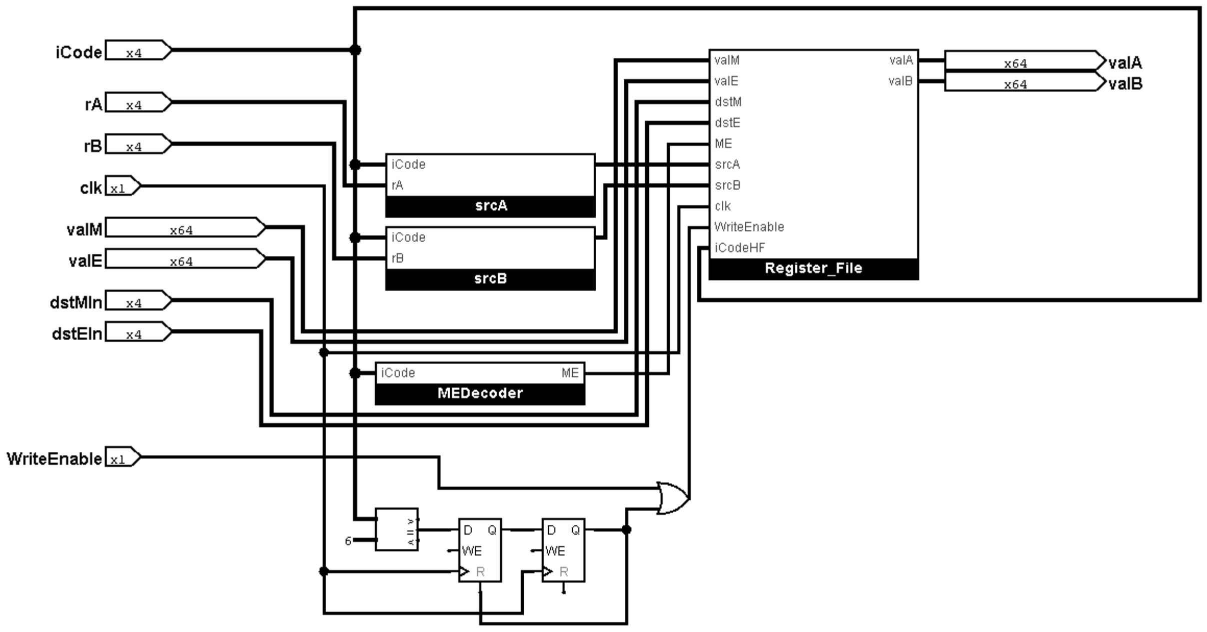

Figure 2: Decode

Execute

The Execute stage takes `iCode`, `iFun`, `valA`, `valB`, and `valC` along with the clock signal as inputs and outputs `valE`. Additionally, the Sign Flag, Overflow Flag, and the Zero Flag are set during this stage.

`iCode`, `valA`, `valB`, and `valC` are fed into `CTRLogic`, which uses the information regarding the current instruction to send into the ALU the two values on which an operation is to be executed.

`ifun` and `icode` are fed into ALUFun to generate a 2 bit output, `noCodenoFun`, to communicate to the ALU which operation to execute.

The ALU takes in 3 inputs, `A`, `B`, and `notFun`, and computes the necessary operation (between addition, subtraction, logical AND, and logical XOR). The output is sent out to `ValE` and stored in a register, and the flags are set in `pcc`.

setCC takes in `iCode` in order to determine if the flag should be set on this operation or not. This output, along with both the clock and the flag outputs, are fed into CC in order to set the flag. This output is then used to set the jump condition in combination with `ifun`, to be used later in PCUpdate.

Contributors

Arjun Kurkal

Screenshots

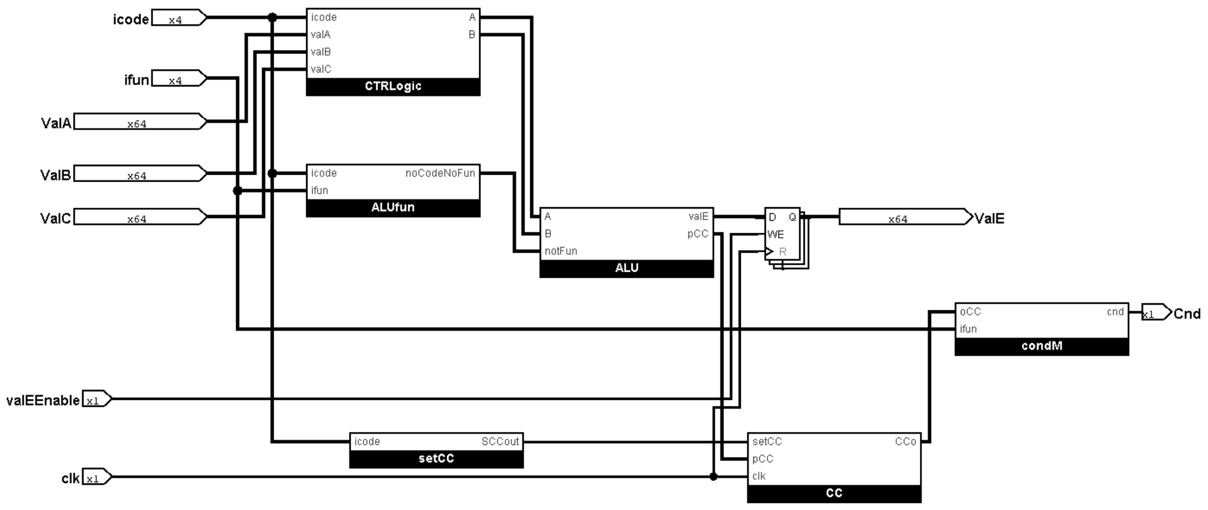

Figure 3: Execute

Memory

The Memory Stage take `iCode`, `valE`, `valA`, and `valP` as well as the clock signal and outputs `valM`. Additionally, when writing value into memory or reading values from memory, a `complete` pulse is send when the Memory Stage is doing doing so.

Icode is fed into MemEnabler, which outputs `oEnable`, `wEnable`, `rEnable`, `addrEnable`, and `datEnable`, which respectively communicate if any output is to be sent to ValM, if any value is to be written to RAM, if any value is to be read from RAM, whether the address for RAM access comes from `valE` or `valA`, and whether or not the write address comes from `valP` or `valA`.

This data is written to or read from RAM using a loop to read and write data byte for byte, which is handled with a counter that starts when the pulse enters the Memory stage. When that pulse reaches the end of memory, it signals to the next instruction that the memory stage is complete.

Contributors

Arjun Kural
Huy Lai

Screenshots

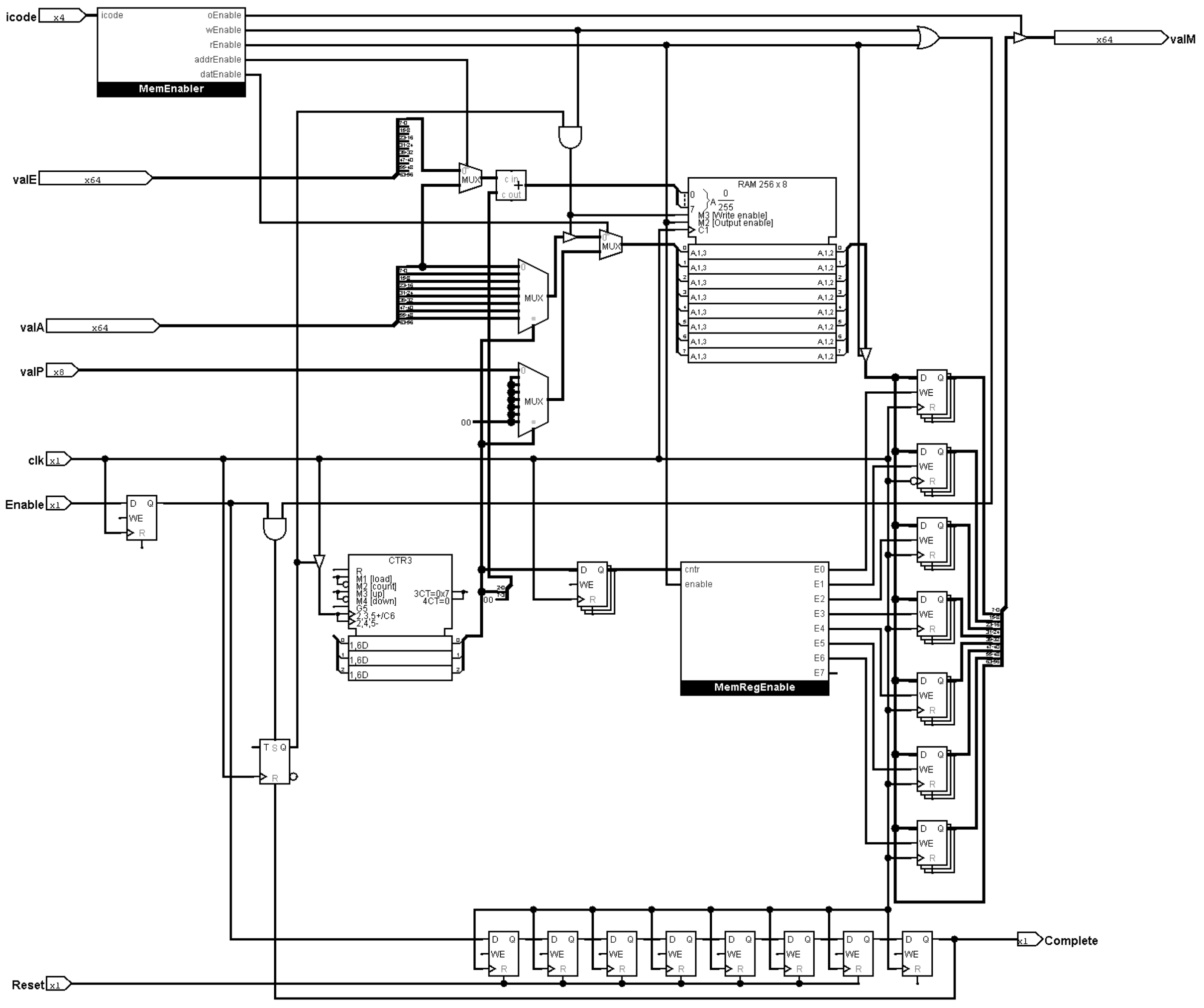

Figure 4: Memory

Write Back

The Write Back stage will allow for the writing to the register file. It takes `iCode`, `rA`, and `rB` as inputs and outputs `dstE` and `dstM`.

Contributors

Huy Lai
Arjun Kurkal

Screenshots

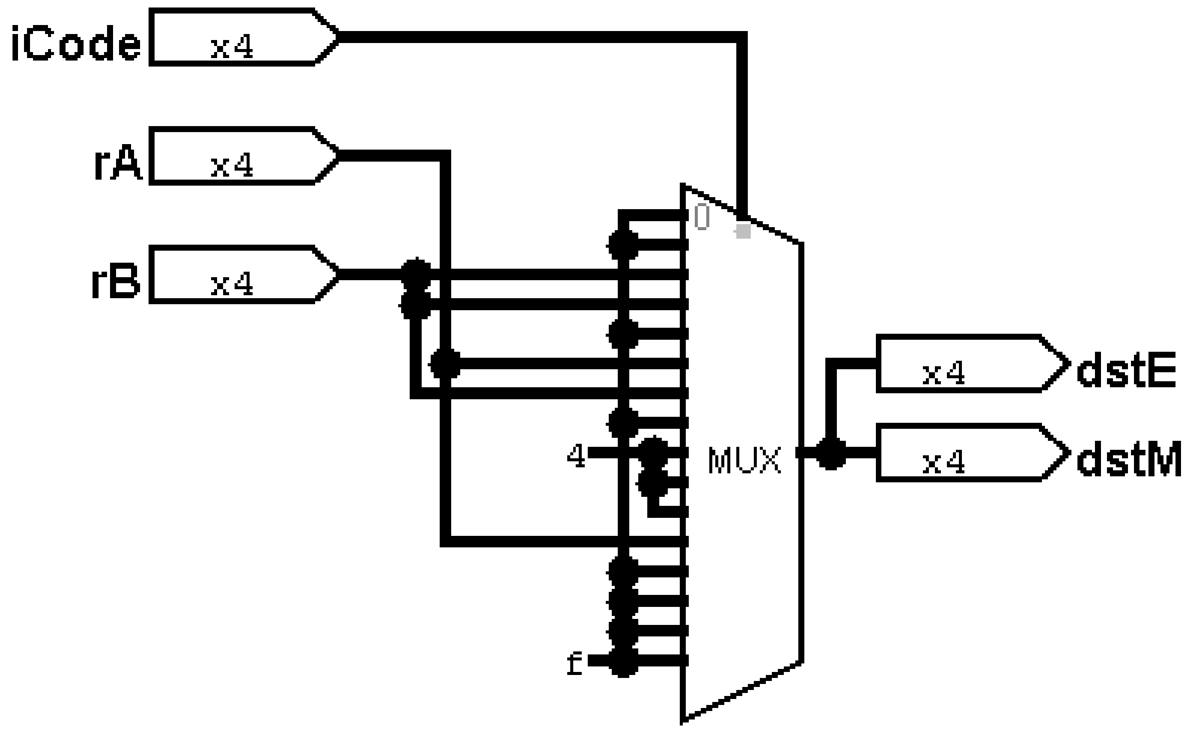

Figure 5: Write Back

PC Update

The PC Update stage will tell the PC where to go next. It takes `iCode`, `valP`, `valM`, `valC` along with `cnd` from the Execute stage as inputs. `cnd` is used on `jXX`, when high this indicates that a jump is being made.

Contributors

Huy Lai
Arjun Kurkal
Kyle Crusius

Screenshots

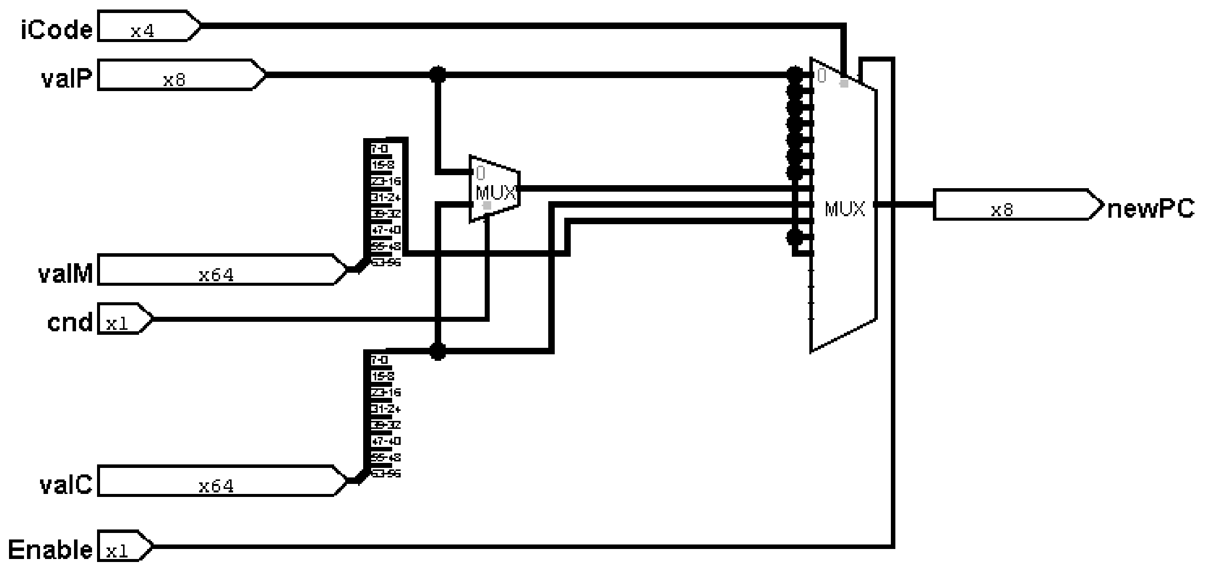

Figure 6: PC Update

Timing Diagrams

Because the timing diagrams are so long, they are split across multiple screen shots.
However each row of the image is always the same signal.
From top to bottom, there are:

- `Clock`
- `iCode`
- `iFun`
- `rA`
- `rB`
- `valC`
- `valP`
- `valA`
- `valB`
- `valE`
- `valM`
- `(new) PC`

Call Test

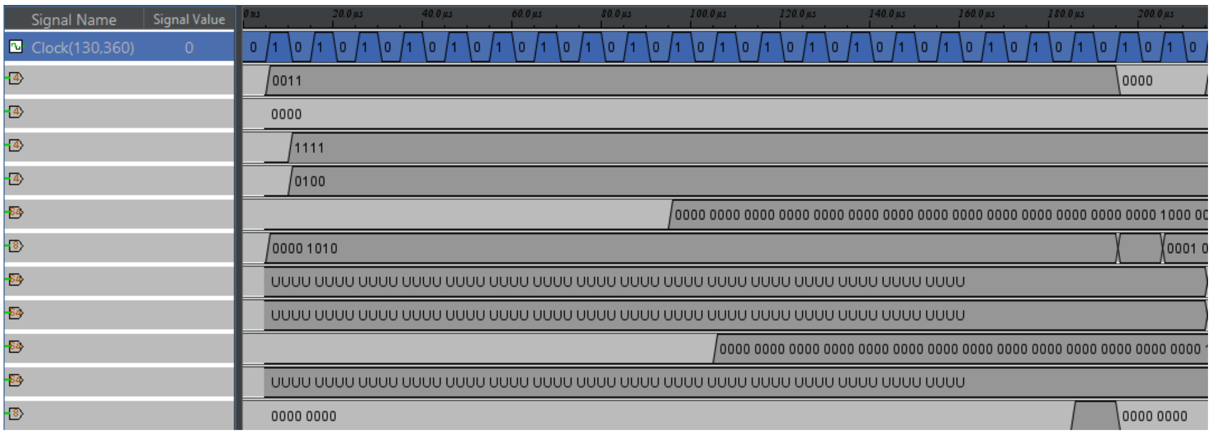
Signal Name
Signal Value
Clock(130,360)
0
0011
0000
0000
1111
0100
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1000
0000 1010
0001
UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU
UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU
0000 0000
0000 0000

Figure 7: Call Test Part 1

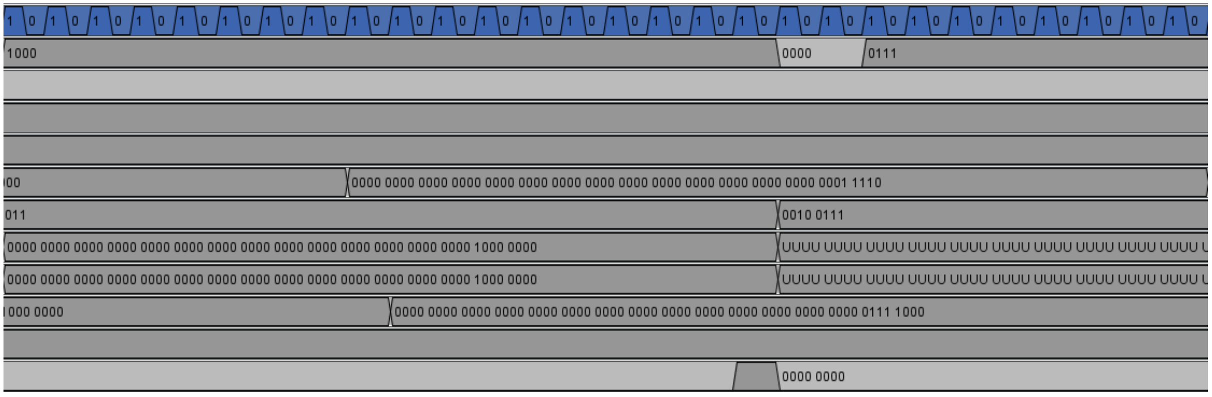

Figure 8: Call Test Part 2

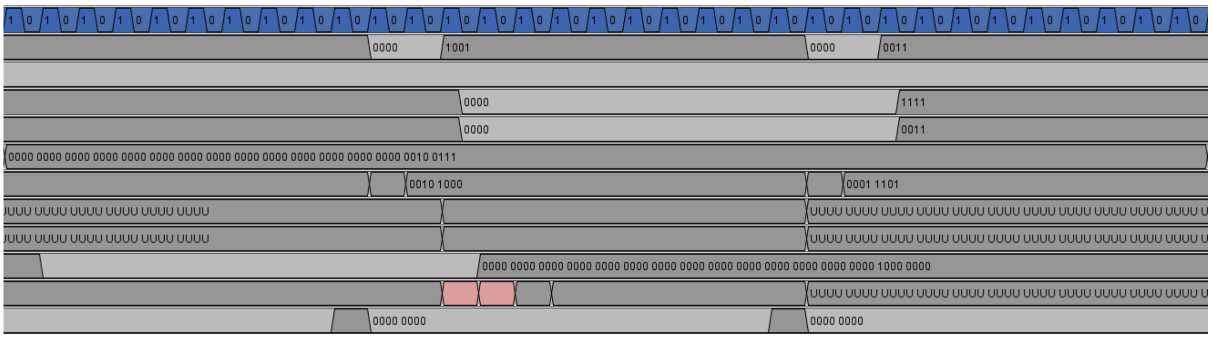

Figure 9: Call Test Part 3

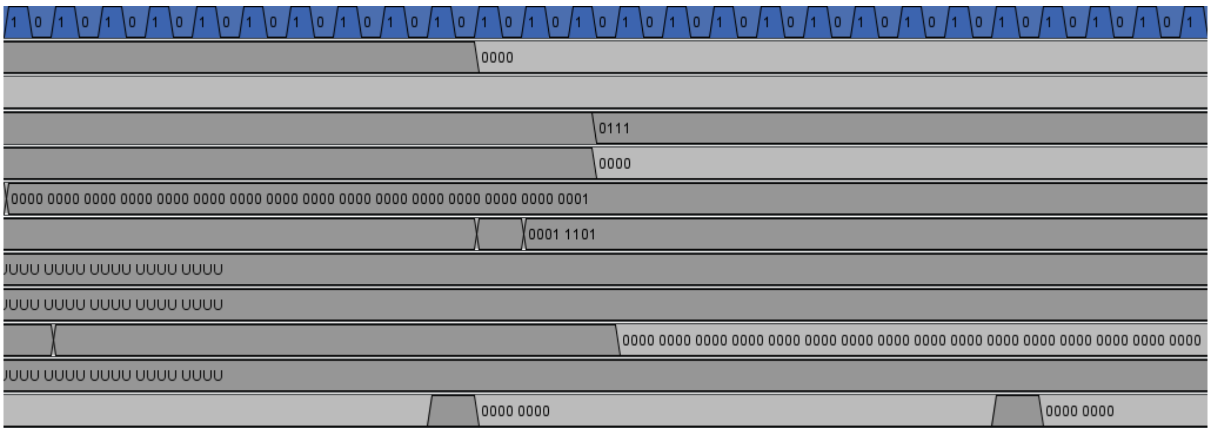

Figure 10: Call Test Part 4

OP Test

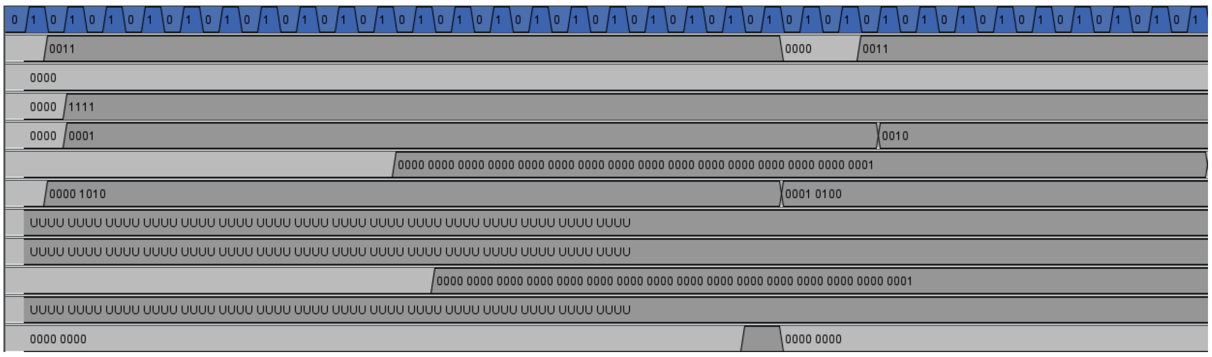
0011
0000
0011
0000
0000
1111
0000
0001
0010
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0001
0000 1010
0001 0100
UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU
UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0001
UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU
0000 0000
0000 0000

Figure 11: OP Test Part 1

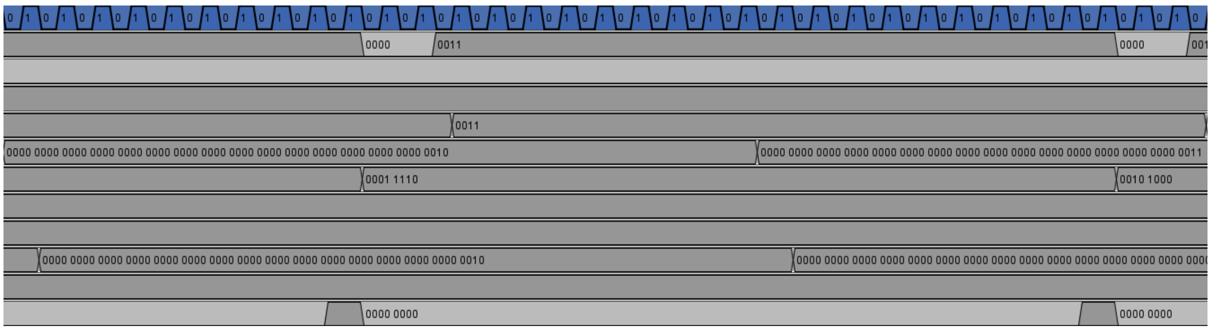

Figure 12: OP Test Part 2

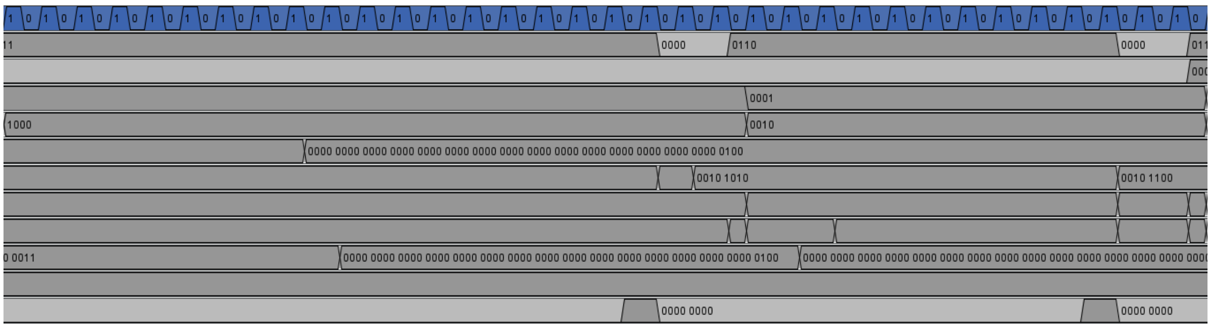

Figure 13: OP Test Part 3

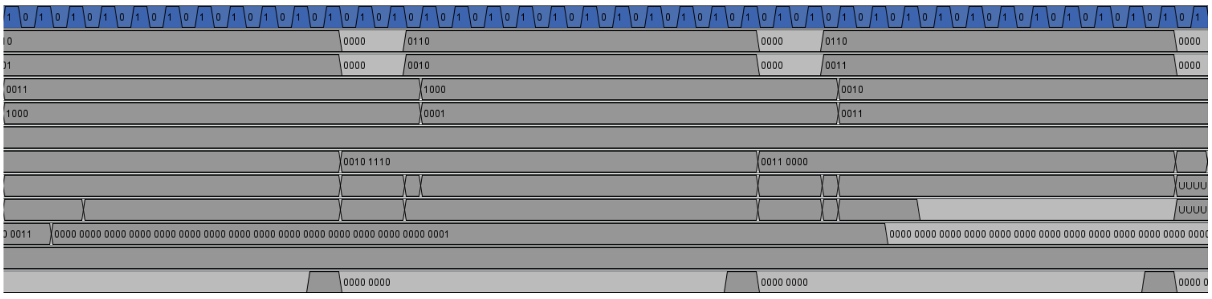

Figure 14: OP Test Part 4

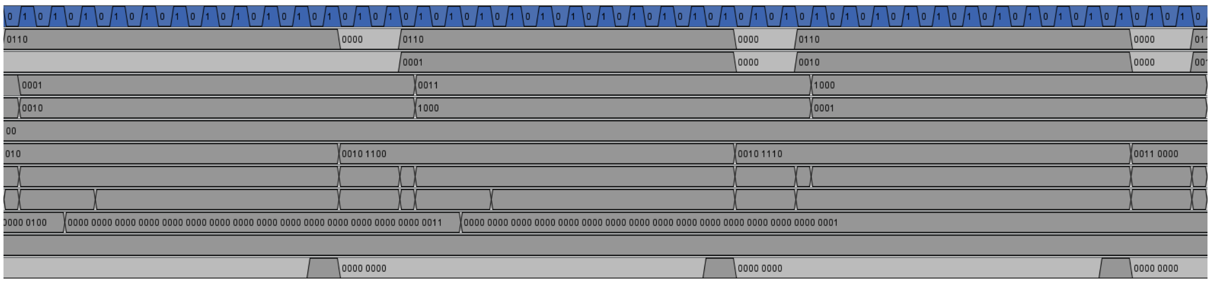

Figure 15: OP Test Part 5

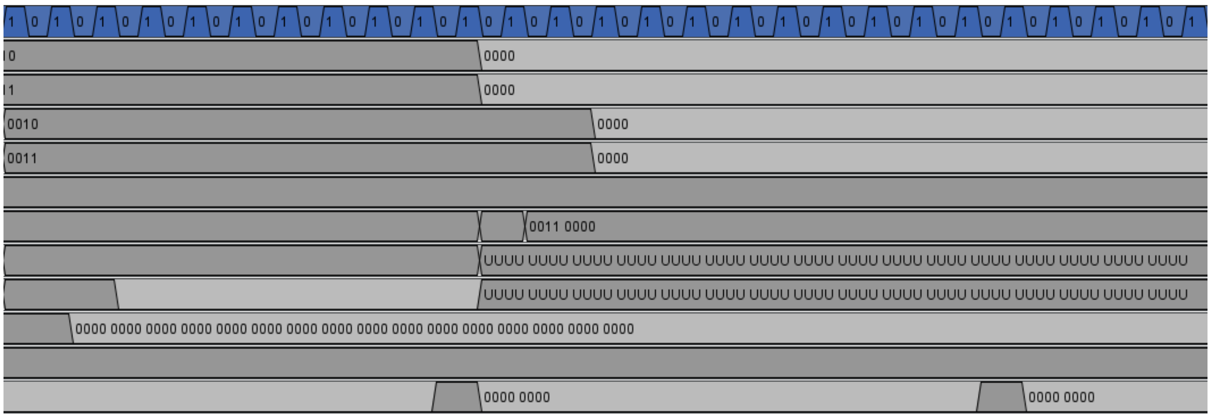

Figure 16: OP Test Part 6

Push Pop Test

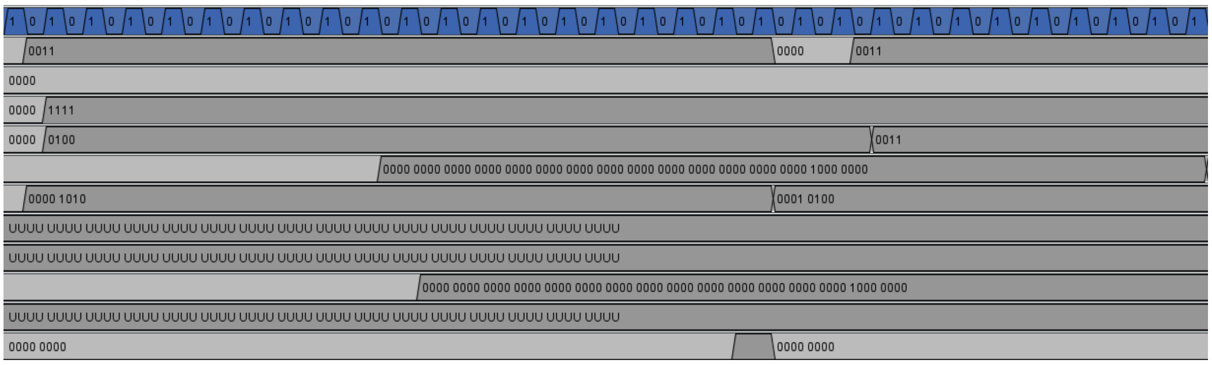
0011
0000
0011
0000
0000
1111
0000
0100
0011
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1000 0000
0000 1010
0001 0100
UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU
UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1000 0000
UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU
0000 0000
0000 0000

Figure 17: Push Pop Test Part 1

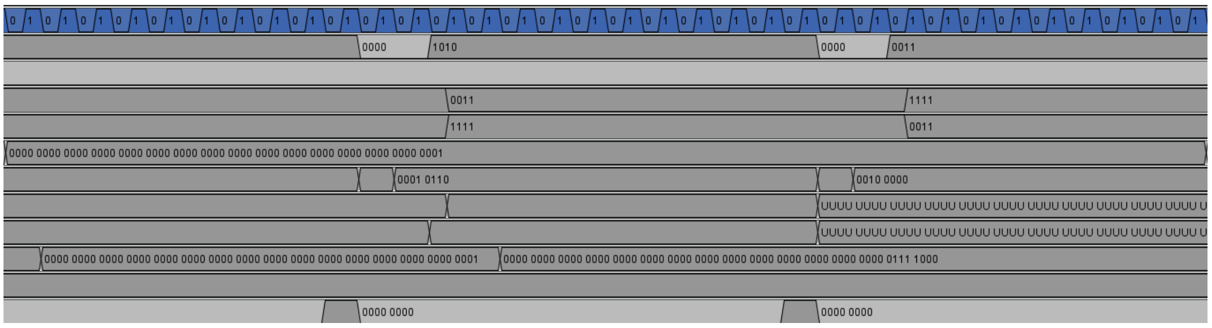

Figure 18: Push Pop Test Part 1

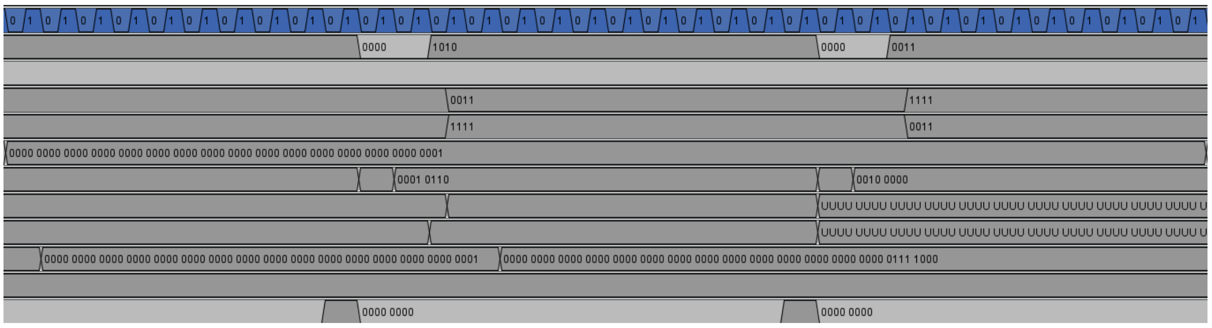

Figure 19: Push Pop Test Part 1

Register Test

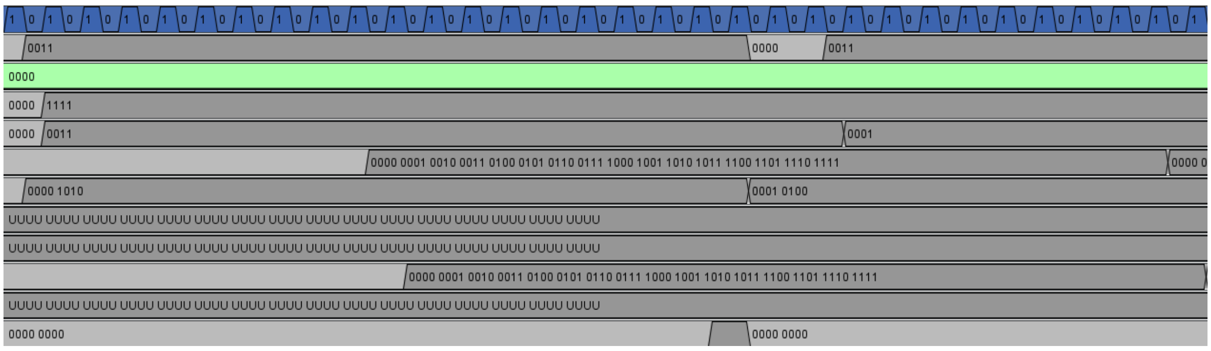
0011
0000
0011
0000
0000
1111
0000
0011
0001
0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
0000 1010
0001 0100
UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU
UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU
0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU UUUU
0000 0000
0000 0000

Figure 20: Register Test Part 1

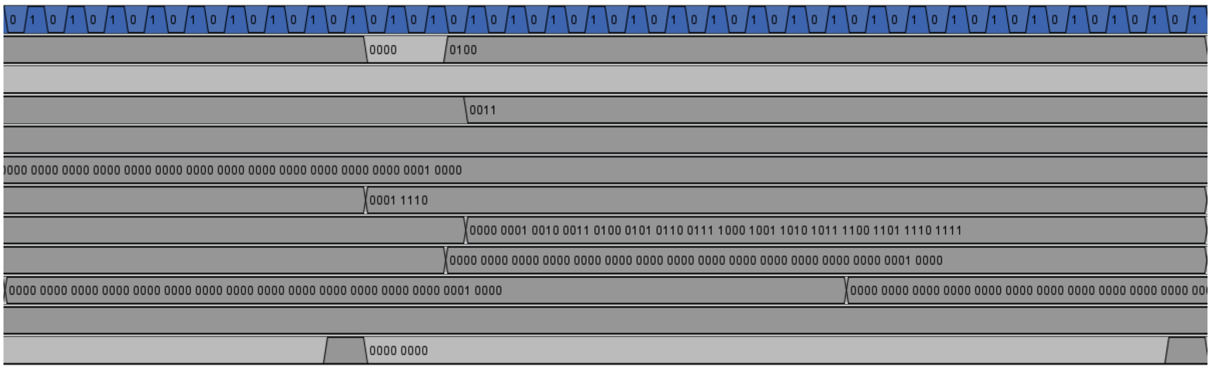

Figure 21: Register Test Part 2

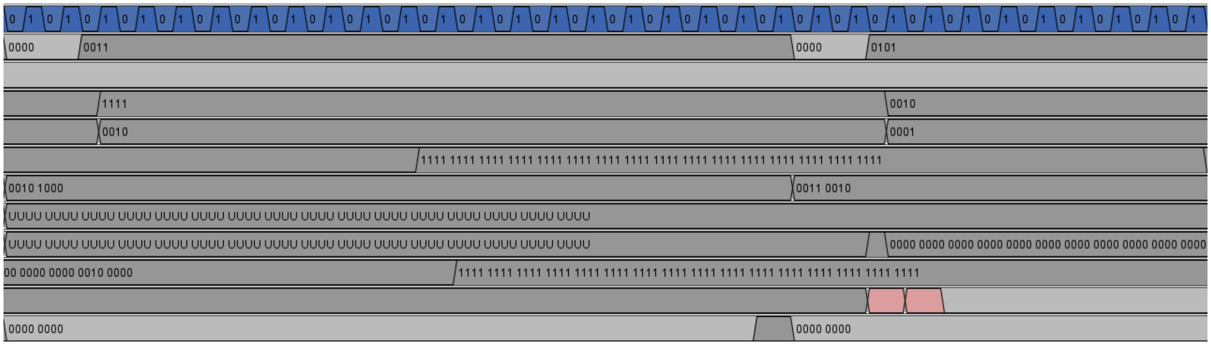

Figure 22: Register Test Part 3

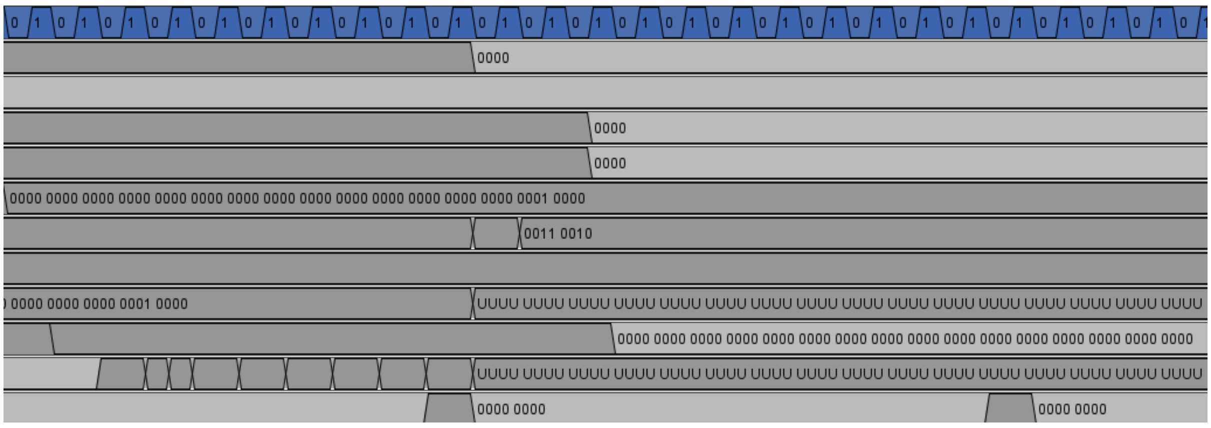

Figure 23: Register Test Part 4