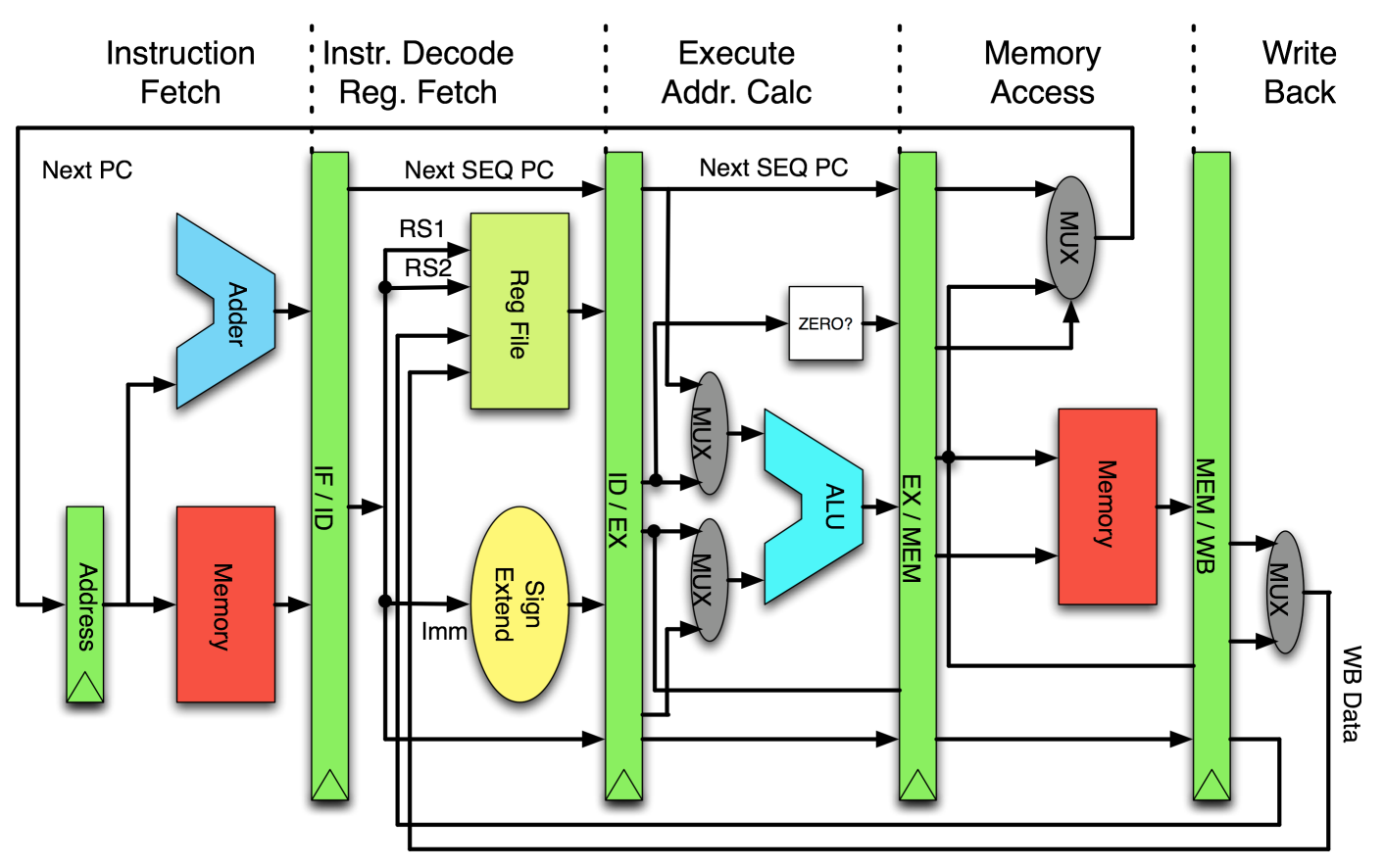
**Computer Architecture Project Description**

It is required to implement using VHDL a simple **Secure** **five stage** **pipelined processor**, Mini-MIPS, which is a subset of the 32-bit MIPS architecture as described in the text book. Mini-MIPS uses the same 3 instruction formats of MIPS (**R, I and J-types**) to implement following 10 instructions:  **ADD, SUB, ADDI, SUBI, AND, OR, LW, SW, JR, and BEQZ**.

**Mini-MIPS Specifications:**

* The Mini-MIPS is 5-stage pipeline: Instruction Fetch, Instruction Decode, Execution, Memory Access, Write Back.
* Only two instructions can access the memory (**LW,SW**).
* The others work on registers are 32 bits large.
* The processor contains 32 registers.
* Register R0 is always 0.
* Data hazards are resolved thanks to Forwarding unit.
* Branch hazards are resolved by predicting the address results.
* It will be assumed that the memory can be accessed in one clock cycle and works synchronously with the CPU (i.e. no need to provide memory control)

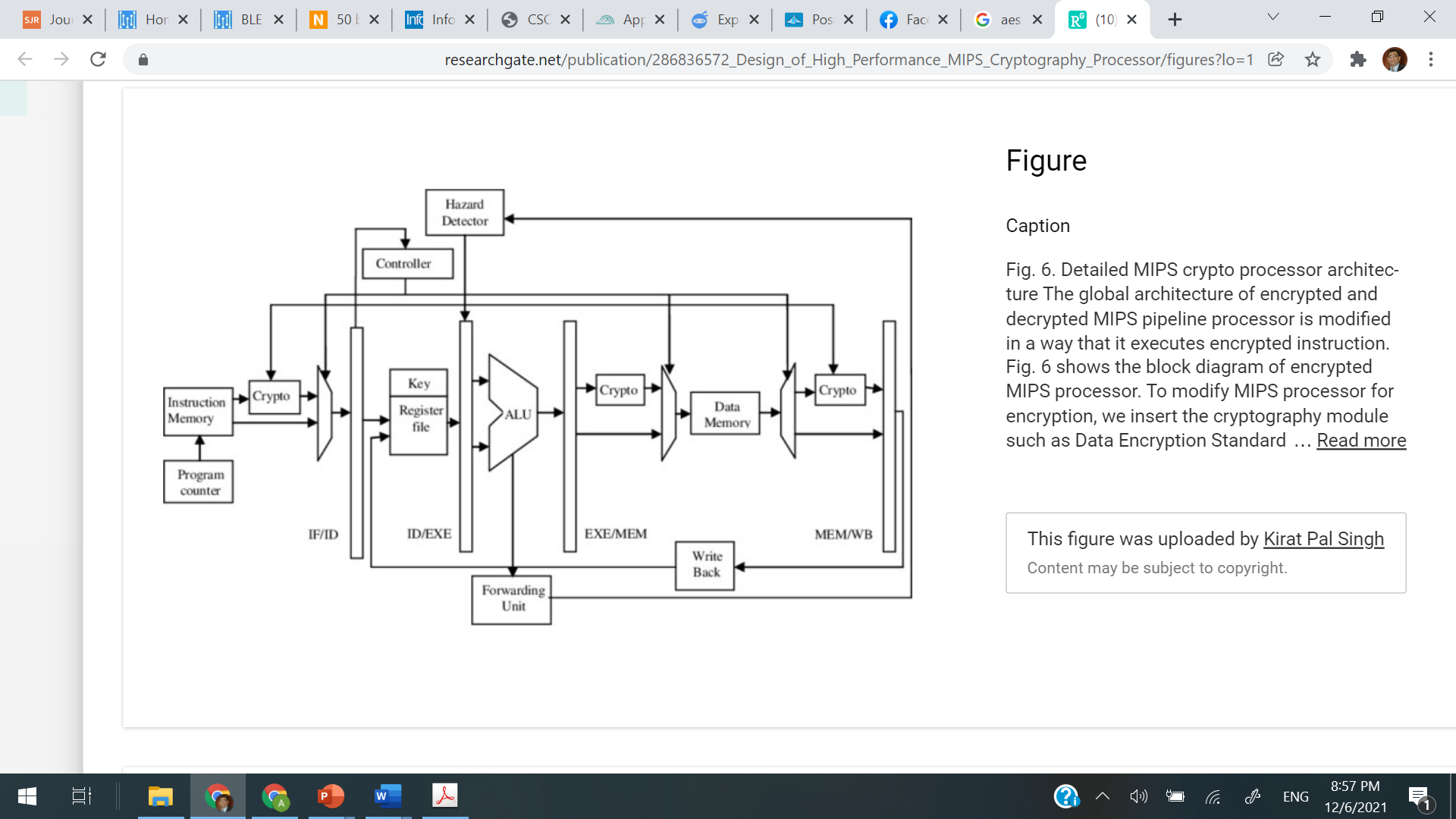


The project is composed of 3 stages

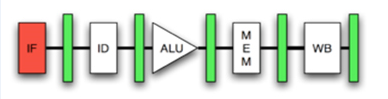
Stage 1: A complete 5 stage processor without Pipelining

Stage 2: A complete 5 stage processor with pipelining and forwarding included

Stage 3: Use an AES Encryption library to encrypt the memory as shown in the figure below.



project Extra Explanation



# Instruction Format

Instruction Fetch stage we read the instruction from program memory and copy it to Instruction Register.

Each memory line is 21 bits

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Opcode | I/R indicator | First argument | Second argument | Third argument | |
| 3 bits | 1 bit | 5 bits | 5 bits | | 5 bits |

Example:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ADD | R1 | | R2 | | R3 |
| 000 | 0 | 00001 | | 000010 | 00011 |

## Opcode assignment

|  |  |  |
| --- | --- | --- |
| Instruction | Opcode | I/R indicator |
| ADD | 000 | 0 |
| SUB | 001 | 0 |
| AND | 010 | 1 |
| OR | 011 | 1 |
| LW | 100 | Do not care |
| SW | 101 | Do not care |
| JR | 110 | Do not care |
| BEQZ | 111 | Do not care |

## Instruction functions & Example program memory entries

|  |  |
| --- | --- |
| ADD R1, R2, R3 | R2 + R3 -> R1 |
| 000 0 00001 00010 00011 | |
| SUB R1, R2, R3 | R2 – R3 -> R1 |
| 001 0 00001 00010 00011 | |
| ADDI R1, R2, 5 | R2 + 5 -> R1 |
| 000 1 00001 00010 00011 | |
| SUBI R1, R2, 5 | R2 – 5 -> R1 |
| 001 1 00001 00010 00011 | |
| AND R1, R2, R3 | R2 and R3 -> R1 |
| 010 X 00001 00010 00011 | |
| OR R1, R2, R3 | R2 or R3 -> R1 |
| 011 X 00001 00010 00011 | |
| LW R1, 5, R2 | Data inside address( R2 + 5) -> R1 |
| 100 X 00001 00101 00010 | |
| SW R1, 5, R2 | R1 -> To address (R2 + 5) |
| 101 X 00001 00101 00010 | |
| JR R1, 5 | R1 + 5 -> Program Counter (PC) |
| 110 X 00001 00101 XXXXX | |
| BEQZ R1, 5, R2 | if( R1 = (others->’0’) then  {  R2+5 -> Program Counter (PC)  } |
| 111 X 00001 00101 00010 | |

# Register File

Register file contains 32 Registers (R0, R1, R2 ….. R31), each Register is 32 bits.

|  |  |
| --- | --- |
| Address | Register |
| 00000 | R0 |
| 00001 | R1 |
| 00010 | R2 |
| \\\\\\\\\\\\\\\\\\\\\\\\\  \\\\\\\\\\\\\\\\\\\\\\\\  \\\\\\\\\\\\\\\\\\\\\\\\\  \\\\\\\\\\\\\\\\\\\\\\\\\  \\\\\\\\\\\\\\\\\\\\\\\\  \\\\\\\\\\\\\\\\\\\\\\\\\ | **\\\\\\\\\\\\\\\\\\\\\\\\\**  **\\\\\\\\\\\\\\\\\\\\\\\\**  **\\\\\\\\\\\\\\\\\\\\\\\\\**  **\\\\\\\\\\\\\\\\\\\\\\\\\**  **\\\\\\\\\\\\\\\\\\\\\\\\**  **\\\\\\\\\\\\\\\\\\\\\\\\\** |
| 11111 | R31 |

R0 is always equal to Zero

## Program memory VS Register file

Program memory contain program opcodes and argument and data saved to memory (Van-Neumann architecture)

Register File is a group of registers used by processor during program execution for fast data access and temporary storage.

It is better to save each instruction output to a register rather than program memory, While the program output should be saved to the main memory for permanent storage.

# Extra Notes

* The processor is 5 stages pipelined, so the only data hazard will be caused due to RAW (Read After Write dependency). This is solved using forwarding and stalling as discussed in both Lecture and tutorial.