PART A : HACK CPU

PART B : CARRY SELECT ADDER

A Project Report

Submitted by

GROUP : 9

Group Members

1. DEV BALA SARAGESH - CB.SC.U4AIE23022

2. GHANASREE S - CB.SC.U4AIE23028

3. HARI HEMAN V K - CB.SC.U4AIE23029

4. RAGHAV N - CB.SC.U4AIE23059

As a part of the subject

22AIE102 – Elements of Computing Systems – 1

in partial fulfilment for the award of the degree of

BACHELOR OF TECHNOLOGY IN

COMPUTER SCIENCE ENGINEERING – ARTIFICIAL INTELLIGENCE

Centre for Computational Engineering and Networking

AMRITA SCHOOL OF ARTIFICIAL INTELLIGENCE

AMRITA VISHWA VIDYAPEETHAM

COIMBATORE – 641 112 (INDIA)

December 2023

CONTENTS

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to all those who contributed to the success of this project entitled **"HACK CPU AND HACK CARRY SELECT ADDER"**. I am also thankful for the support and guidance provided by **Dr. Jyothish Lal.G**, whose insights and encouragement significantly contributed to the project's development. The collective efforts of the entire team have resulted in an outcome we can all be proud of. Thank you to everyone involved for your unwavering commitment and hard work.

PROJECT TEAM:

1. DEV BALA SARAGESH B S (CB.SC.U4AIE23022)
2. GHANASREE S (CB.SC.U4AIE23028)
3. HARI HAMEN V K (CB.SC.U4AIE23029)
4. RAGHAV N (CB.SC.U4AIE23059)

ABSTRACT

The abstract encapsulates the fundamental principles and advantages of a Carry-Select Adder (CSA) and a Hardware Description Language (HDL) Central Processing Unit (CPU). The Carry-Select Adder (CSA) is a high-speed digital circuit designed for rapid addition of multi-bit numbers by efficiently generating and selecting carry signals. Its parallel structure allows simultaneous computation of multiple carry chains, reducing propagation delay and enhancing performance compared to traditional adders. However, this improvement comes with increased hardware complexity due to duplicated adder modules.

On the other hand, the HDL-based CPU is a versatile design that leverages Hardware Description Language for CPU architecture. It encompasses components like the ALU, control unit, registers, and instruction set architecture (ISA), offering the benefits of simulation and verification before physical implementation. This approach facilitates faster development cycles and allows for tailored designs meeting specific performance, power efficiency, or application needs.

In essence, the abstract highlights the speed enhancements of the Carry-Select Adder (CSA) and the versatility of an HDL-based CPU, showcasing their architectural features, advantages, and potential applications in various computing domains.

PART – A

16-bit HACK CPU

INTRODUCTION

The implementation of a 16-bit Central Processing Unit (CPU) using Hardware Description Language (HDL) marks a significant stride in the realm of digital system design. The evolution of CPUs has been driven by the constant pursuit of enhanced computational capabilities, and leveraging HDLs for CPU design offers a powerful and flexible approach. This introduction provides an overview of the motivation, challenges, and potential benefits associated with the HDL implementation of a 16-bit CPU.

The demand for more sophisticated computing systems capable of handling complex tasks has led to a resurgence of interest in 16-bit CPUs. These processors strike a balance between the simplicity of 8-bit architectures and the computational power of 32-bit architectures, making them suitable for a broad range of applications, including embedded systems, IoT devices, and control systems. The use of HDL in this context is motivated by the need for a systematic and efficient way to describe the intricate hardware components and functionalities of a 16-bit CPU.

HDL implementation of a 16-bit CPU offers numerous advantages. The abstraction provided by HDLs allows designers to express complex hardware functionalities in a manner similar to high-level programming languages, promoting rapid prototyping and design exploration. Furthermore, HDLs enable a more streamlined hardware-software co-design process, facilitating collaboration between hardware and software engineers. The modularity and reusability inherent in HDL-based designs contribute to scalable and maintainable 16-bit CPU implementations.

This endeavour to implement a 16-bit CPU using HDL represents a convergence of digital design principles, computer architecture, and programming language concepts. As we delve into the details of this implementation, we aim to address the aforementioned challenges and capitalize on the benefits, ultimately contributing to the broader landscape of efficient and versatile 16-bit computing systems.

OBJECTIVES

The objectives for the HACK CPU project, as part of the NAND to Tetris course, are focused on guiding participants through the creation of a fully functional computer system. Here are the key objectives for the HACK CPU project:

1. Understanding Computer Architecture: Gain a comprehensive understanding of computer architecture by constructing a central processing unit (CPU) from basic logic gates.
2. Hands-on Learning: Engage in a hands-on, bottom-up learning experience by progressively building components, starting with NAND gates and advancing to more complex units like the ALU (Arithmetic Logic Unit) and memory.
3. Instruction Set Design: Design and implement a simple instruction set for the HACK CPU that is both comprehensible and efficient, facilitating ease of programming.
4. ALU Design: Design and integrate an Arithmetic Logic Unit (ALU) into the CPU, capable of performing basic arithmetic and logic operations.
5. Memory Handling: Implement memory units and understand how the CPU interacts with memory to fetch and store instructions and data.
6. Instruction Execution: Develop a deep understanding of the process of fetching, decoding, and executing instructions within the CPU.
7. Input/Output Handling: Integrate input/output mechanisms into the system, allowing the CPU to interact with external devices.
8. Testing and Debugging: Gain experience in testing and debugging the HACK CPU at each stage of development, ensuring its correct functionality.
9. Programming Language Development: Realize the connection between hardware and software by implementing a basic programming language that the HACK CPU can execute.
10. Documentation and Communication: Document the design choices, circuit schematics, and explanations clearly, fostering effective communication of complex technical concepts.

DESIGN

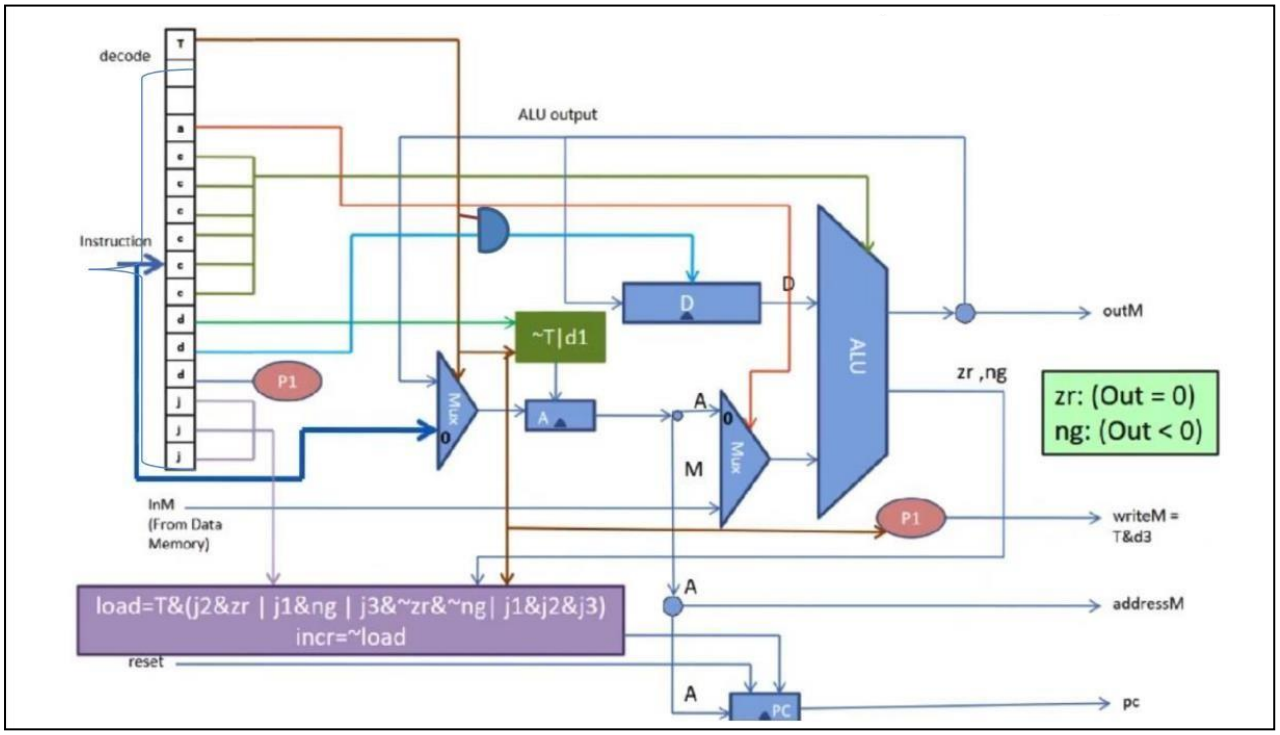


Figure 1: The chip design of the HACK CPU

METHODOLOGY

1. **Inputs and Outputs:**

    1. **Inputs (IN):**

        inM[16]: Input representing the M value (contents of RAM[A]).

        instruction[16]: Input representing the instruction for execution.

        reset: Input which resets the PC

    2. **Outputs (OUT):**

        outM[16]: Output representing the M value.

        writeM: Output signaling whether to write into M.

        addressM[15]: Output representing the address in data memory (of M).

        pc[15]: Output representing the address of the next instruction.

2. **Internal Parts:**

    1. **Selecting the input for the ALU:**
        - **Not(in=instruction[15], out=ni):** Negates the 15th bit of the instruction, indicating whether it's an A or C instruction.
        - **Mux16(a=outtM, b=instruction, sel=ni, out=i):** Selects the input to be loaded into the A register based on the type of instruction.

    2. **Loading A Register:**
        - **Or(a=ni, b=instruction[5], out=intoA):** Determines whether to load the A register.
        - **Register(in=i, load=intoA, out=A, out[0..14]=addressM):** Loads the input based on the load output.

    3. **Selecting A or M input for ALU:**
        - **And(a=instruction[15], b=instruction[12], out=AorM):** Determines the selection line for the MUX to choose between A or M input.
        - **Mux16(a=A, b=inM, sel=AorM, out=AM):** Selects the A or M input based on the load.

    4. **ALU Operation:**

    ALU(x=D, y=AM, zx=instruction[11], nx=instruction[10], zy=instruction[9], ny=instruction[8], f=instruction[7], no=instruction[6], out=outtM, out=outM, zr=zr, ng=ng): Performs ALU operation based on control bits and inputs.

5. **Loading D Register:**
   - **And(a=instruction[15], b=instruction[4], out=intoD):** Determines whether to load the D register.
   - **Register(in=outtM, load=intoD, out=D):** Loads the output of the ALU if it is a C instruction and the d2 bit is 1.

6. **Determining writeM Bit:**
   - **And(a=instruction[15], b=instruction[3], out=writeM):** Determines the writeM bit based on the C instruction and the d3 bit.

7. **Load bit of the PC for the jump case:**

   Determines conditions for jumping:

   - **Not(in=ng, out=pos), Not(in=zr, out=nzr):** Negates the negative and zero flags.

     Various And gates are used to combine conditions for different jump scenarios. Or gates combine the conditions to determine the load bit for the program counter.

8. **Program Counter (PC):**
   - **PC(in=A, load=ld, inc=true, reset=reset, out[0..14]=pc):** The program counter is driven by conditions for jumping, incrementing the address value, or resetting the CPU.

RESULT

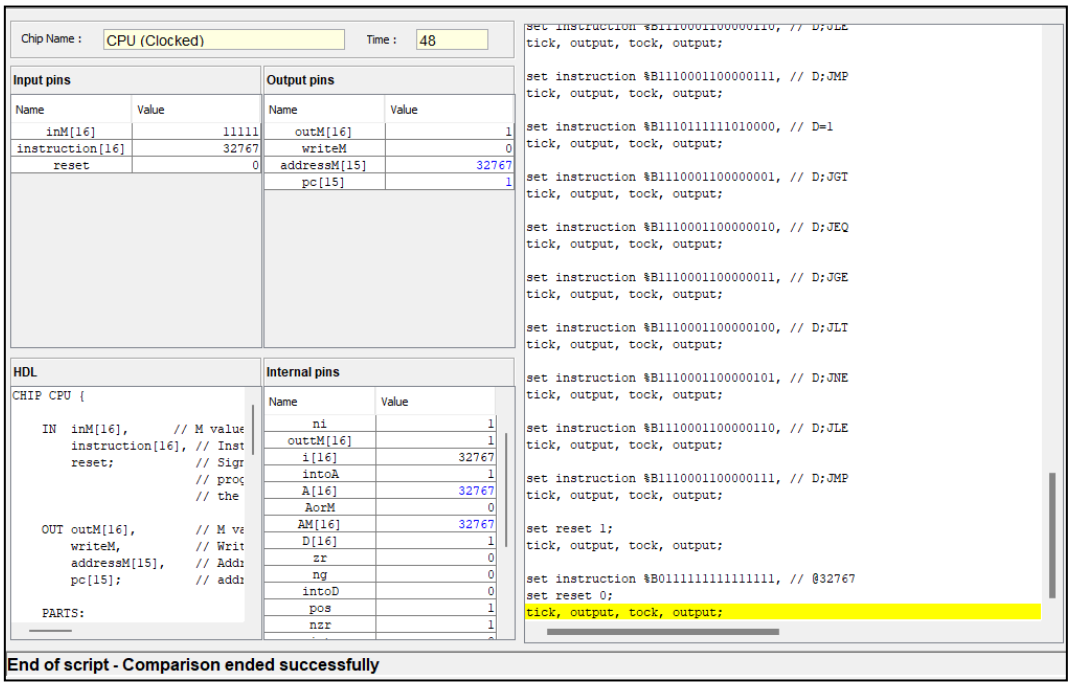


Figure 2: The output of the CPU in the Hardware simulator tested with the test script.

APPLICATIONS

A 16-bit CPU implemented using Hardware Description Language (HDL) can find applications in various computing systems, ranging from embedded systems to more complex computational tasks. Here are some potential applications for a 16-bit HDL-based CPU:

- Embedded Systems
- Educational Purposes
- Retro Computing
- Custom Computing Solutions
- Prototyping and Research
- Low-Power Applications
- Control Systems
- Simulation and Verification:

Overall, the application of a 16-bit HDL-based CPU spans a wide range of domains, offering versatility, customization, and adaptability to specific computing requirements.

PART – B

16- CARRY SELECT ADDER

INTRODUCTION

A Carry Select Adder (CSA) is a digital circuit that performs the addition of two binary numbers. It is designed to improve the speed of addition by concurrently generating multiple carry chains and selecting the correct result based on the actual carry input. This project is often undertaken in digital design and computer architecture courses to deepen understanding of arithmetic circuits and optimize the performance of adders.

The Carry Select Adder operates by dividing the addition process into two stages: the generation of two separate sum outputs assuming either a carry of 0 or 1, and the subsequent selection of the correct sum based on the actual carry input. This approach enables parallel processing of the potential results, contributing to faster overall addition times.

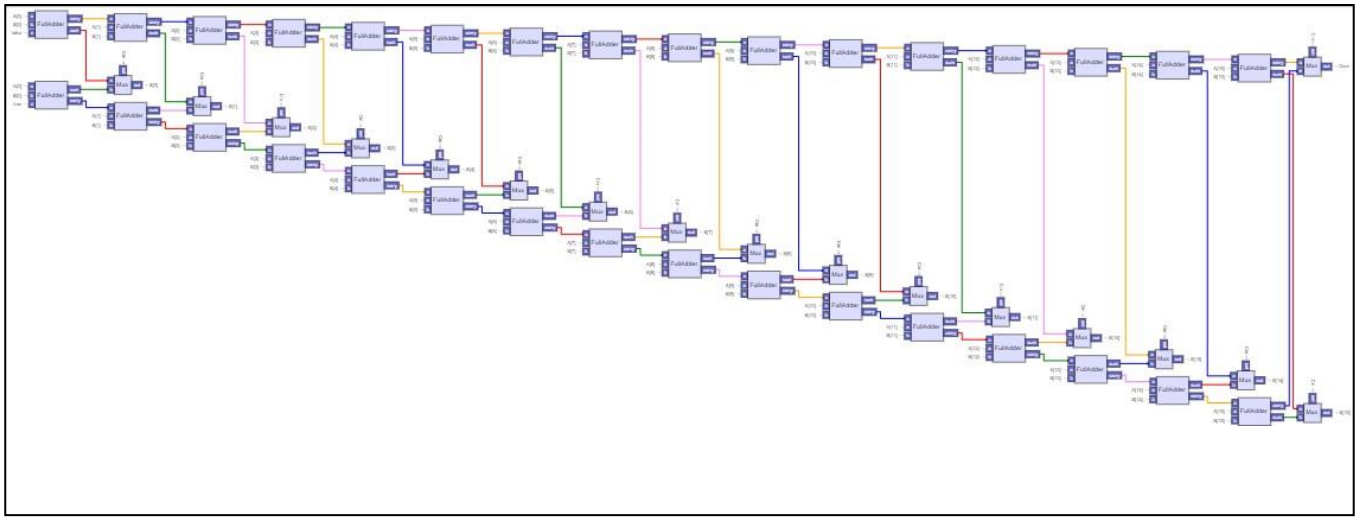
The project involves designing and implementing a Carry Select Adder using basic digital logic gates, multiplexers, and other standard components. Participants typically explore different design strategies, analyze trade-offs between speed and complexity, and gain hands-on experience in optimizing arithmetic circuits.

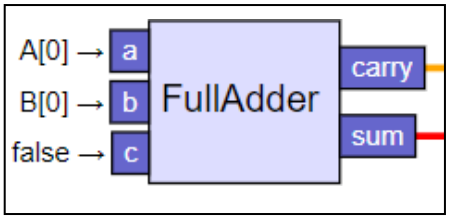
Through the Carry Select Adder project, participants deepen their understanding of digital logic design principles, gain insights into the challenges of optimizing arithmetic circuits for speed and efficiency, and acquire practical skills in implementing complex circuits. This project is a valuable addition to the curriculum for those studying digital electronics, computer architecture, and related fields, providing a concrete application of theoretical concepts and fostering a hands-on approach to learning.

OBJECTIVES

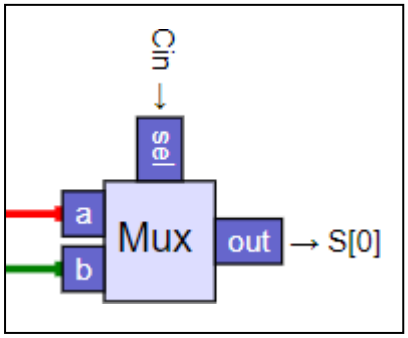
1. **Understanding Binary Addition:** Gain a solid understanding of binary addition and the basic principles of arithmetic circuits.
2. **Applying Digital Logic Design Concepts:** Apply digital logic design concepts to create a functional Carry Select Adder circuit using basic building blocks such as AND gates, OR gates, XOR gates, and multiplexers.
3. **Optimizing Circuit Performance:** Explore strategies for optimizing the performance of the Carry Select Adder, considering factors such as speed, area, and power consumption.
4. **Analyzing Trade-offs:** Evaluate trade-offs between different design choices, such as the size of the lookahead adders, the number of multiplexers, and the complexity of the carry lookahead logic.
5. **Implementing Parallelism:** Understand and implement the concept of parallelism in the Carry Select Adder, allowing for simultaneous processing of potential sum outputs based on different carry inputs.
6. **Verifying Circuit Correctness:** Develop skills in verifying the correctness of the Carry Select Adder circuit through simulation tools, testing, and debugging.
7. **Documenting the Design:** Create clear and concise documentation that outlines the design choices, circuit schematics, and explanations of the functioning of the Carry Select Adder.
8. **Comparing with Other Adder Architectures:** Compare the performance of the Carry Select Adder with other adder architectures, such as Ripple Carry Adders or Carry Lookahead Adders, in terms of speed and resource utilization.
9. **Real-world Applications:** Understand real-world applications where the use of a Carry Select Adder might be advantageous and explore scenarios where it might be preferred over other adder designs.
10. **Hands-on Experience:** Gain practical, hands-on experience in designing, simulating, and implementing digital circuits, reinforcing theoretical concepts learned in digital design courses.

DESIGN





X 32



X 17

METHODOLOGY

16-bit Carry Select Adder (CSA), a digital circuit used for adding two 16-bit binary numbers. The CSA operates with an additional input called Cin (carry-in) and produces two outputs: S (the sum of A and B) and Cout (carry-out). The CSA architecture is implemented using Full Adders and Multiplexers.

Full Adders for Main Addition:

- Sixteen instances of the FullAdder chip are used for the main addition of corresponding bits in A and B (bits 0 to 15).
- Each Full Adder produces a sum (s0 to s15) and a carry-out (c0 to c15).

Full Adders for Overflow Case:

- Another set of sixteen FullAdder instances is used for the overflow addition, considering a carry-in of 1 for each bit (bits 0 to 15).
- This produces additional sums (s16 to s31) and carry-outs (c16 to c31).

Multiplexers for Carry-Out (Cout):

- A Mux is used to select the carry-out for the entire operation. The choice between the carry-out of the main addition (c15) and the overflow addition (c31) is based on the value of the carry-in (Cin).

Multiplexers for Sum (S):

- Sixteen Mux instances are used to select the sum of each bit based on the carry-in (Cin). The choice is between the sum of the main addition (s0 to s15) and the overflow addition (s16 to s31).

In summary, the code efficiently implements a 16-bit Carry Select Adder, considering both the main addition and the overflow case. The use of Full Adders and Multiplexers allows for a modular and scalable design, making it suitable for integration into larger digital systems.

RESULTS

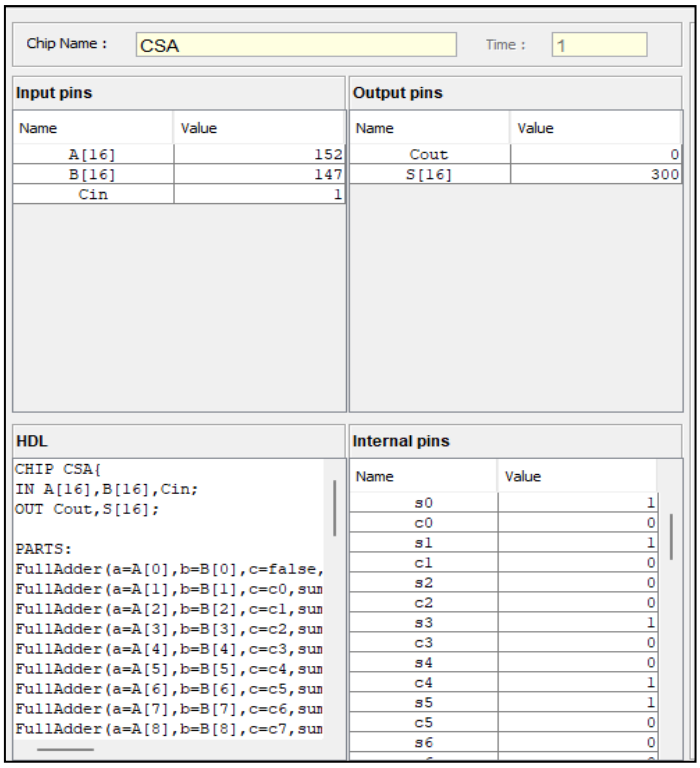


Figure 3: Inputted 152, 147, 1 and output is 300.

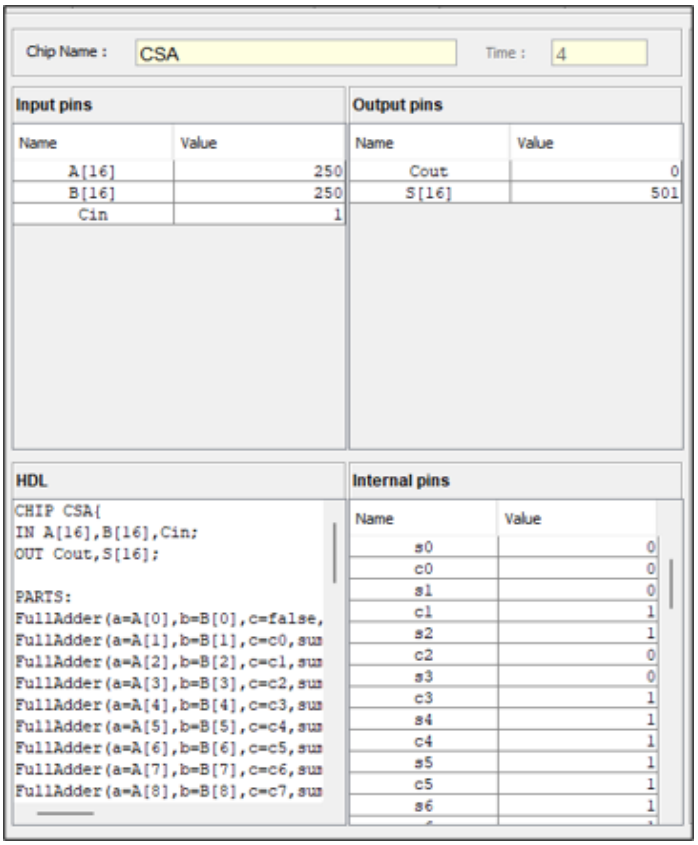


Figure 4: Input is 250,250,1 and output is 501

APPLICATIONS

A Carry Select Adder (CSA) is a digital circuit used for the addition of multi-bit binary numbers. It provides a faster alternative to ripple-carry adders by exploiting parallelism. Here are some applications and advantages of Carry Select Adders:

1. **Arithmetic Units in Processors:**
    - CSA is commonly used in the arithmetic units of microprocessors and digital signal processors (DSPs) to perform addition operations efficiently.

2. **Parallel Prefix Adders:**
    - CSA is a fundamental component in the construction of parallel prefix adders, such as the Brent-Kung adder or the Kogge-Stone adder. These adders are designed for high-speed addition in parallel computing architectures.

3. **High-Performance Computing:**
    - In applications where high-speed arithmetic operations are critical, such as in scientific computing or graphics processing units (GPUs), CSA can be employed to accelerate addition operations.

4. **Digital Signal Processing (DSP):**
    - CSA is well-suited for applications in DSP where rapid signal processing is required, including audio and video processing, telecommunications, and image processing.

5. **Fixed-Point Arithmetic:**
    - In digital systems that use fixed-point arithmetic, CSA can be used for efficient addition of fixed-point numbers, ensuring accurate results.

6. **Addition of Large Binary Numbers:**
    - When adding large binary numbers, ripple-carry adders may introduce significant delays due to the propagation of carry bits. CSA provides a parallelized approach, reducing the overall addition time.

7. **Energy-Efficient Designs:**
    - In scenarios where power efficiency is crucial, CSA can be a favorable choice as it enables faster addition with reduced power consumption compared to ripple-carry adders.

In summary, the Carry Select Adder is a versatile component used in various digital systems and applications where fast and efficient binary addition is required. Its parallelized structure makes it suitable for high-performance computing and applications demanding rapid arithmetic operations.

REFERENCES

1. https://github.com/havivha/Nand2Tetris/blob/master/05/CPU.hdl

2. https://www.nand2tetris.org/project05

3. https://people.duke.edu/~nts9/logicgates/CPU.hdl

4. The Elements of Computing Systems
   Building a Modern Computer from First Principles
   By Noam Nisan, Shimon Schocken · 2005

CONCLUSION

In the realm of digital design, the Carry Select Adder (CSA) project and the HACK CPU project represent immersive educational experiences that delve into the intricacies of circuit optimization and computer architecture, respectively. The Carry Select Adder project serves as a crucible for understanding binary addition and arithmetic circuitry optimization. By breaking down the addition process into parallel stages, participants gain a profound grasp of the trade-offs involved in optimizing for speed and complexity. Through hands-on implementation of digital logic design concepts, the project facilitates the exploration of various design strategies and the critical evaluation of their implications on performance metrics. Beyond the technical aspects, participants cultivate essential skills in verifying circuit correctness, testing, debugging, and documenting their designs. The Carry Select Adder project thus stands as a testament to the fusion of theoretical knowledge with practical application, preparing participants to navigate the complexities of digital circuit design.

On the other end of the spectrum, the HACK CPU project, a pivotal component of the NAND to Tetris course, propels participants into the heart of computer architecture and digital design. This project unfolds as a journey through abstraction layers, guiding learners from the foundational building blocks of NAND gates to the assembly of a fully operational central processing unit. The HACK CPU project not only imparts a comprehensive understanding of instruction set design, Harvard architecture, and the functioning of the Arithmetic Logic Unit (ALU) but also emphasizes the symbiotic relationship between hardware and software. As participants construct memory units, implement input/output mechanisms, and execute instructions within the CPU, they gain practical insights into the complex interplay that defines computing systems. This hands-on exploration fosters critical thinking and problem-solving skills, preparing participants for the dynamic challenges inherent in digital design and computer architecture.

In essence, both the Carry Select Adder and HACK CPU projects embody the ethos of experiential learning in the realm of digital design. These projects go beyond theoretical concepts, offering participants the opportunity to engage with the material in a tangible and transformative way. Whether optimizing arithmetic circuits or constructing a fully functional CPU, participants in these projects emerge not only with technical proficiency but also with a deep appreciation for the multifaceted nature of digital systems. Through the synthesis of theoretical knowledge, hands-on application, and critical analysis, these projects contribute to a holistic educational experience, empowering learners to navigate the complexities of contemporary digital design challenges.

APPENDIX

HACK CPU

```
CHIP CPU {

   IN  inM[16],     // M value input  (M = contents of RAM[A])
       instruction[16], // Instruction for execution
       reset;          // Signals whether to re-start the current
                       // program (reset=1) or continue executing
                       // the current program (reset=0).

   OUT outM[16],       // M value output
           writeM,         // Write into M?
           addressM[15],   // Address in data memory (of M)
           pc[15];         // address of next instruction

   PARTS:
           //Selecting the input for the ALU
           Not(in=instruction[15],out=ni);
           //obtaining the information about the type of information (A or C instruction)
           Mux16(a=outtM,b=instruction,sel=ni,out=i);
           //Selecting the input to be loaded into the a register
           Or(a=ni,b=instruction[5],out=intoA);
           //Getting the value of the load
           ARegister(in=i,load=intoA,out=A,out[0..14]=addressM);
           //loading the input based on the load output
           And(a=instruction[15],b=instruction[12],out=AorM);
           //Selection line for the mux to choose where A or M input to be done
```

```
        Mux16(a=A,b=inM,sel=AorM,out=AM);

        //Selecting the A or M input based on the load

        ALU(x=D,y=AM,zx=instruction[11],nx=instruction[10],zy=instruction[9],ny=instruction
[8],f=instruction[7],no=instruction[6],out=outtM,out=outM,zr=zr,ng=ng);

        //From the above operations one of the input is selected and the control bits are
inputed

        And(a=instruction[15],b=instruction[4],out=intoD);

        //Getting the value of the load bit

        DRegister(in=outtM,load=intoD,out=D);

        //Output of the ALU is loaded if it is a C instuction and d2 bit is 1

        And(a=instruction[15],b=instruction[3],out=writeM);

        //If it is a C instruction and the d3 is 1 then the output bit writeM is 1, so that the
output of the ALU is loaded into the Mregister in the RAM

        //Load bit of the PC for the jump case

        Not(in=ng,out=pos);

        Not(in=zr,out=nzr);

        And(a=instruction[15],b=instruction[0],out=jgt);

        And(a=pos,b=nzr,out=posnzr);

        And(a=jgt,b=posnzr,out=ld1);


        And(a=instruction[15],b=instruction[1],out=jeq);

        And(a=jeq,b=zr,out=ld2);


        And(a=instruction[15],b=instruction[2],out=jlt);

        And(a=jlt,b=ng,out=ld3);


        Or(a=ld1,b=ld2,out=ldt);

        Or(a=ld3,b=ldt,out=ld);

//Program counter used in case there is a jump case or increment the address value or reset
the CPU

        PC(in=A,load=ld,inc=true,reset=reset,out[0..14]=pc); }
```

HACK CARRY SELECT ADDER

```
CHIP CSA{
IN A[16],B[16],Cin;
OUT Cout,S[16];
PARTS:
//Adding the two 16 bit number assuming the input carry is 0
FullAdder(a=A[0],b=B[0],c=false,sum=s0,carry=c0);
FullAdder(a=A[1],b=B[1],c=c0,sum=s1,carry=c1);
FullAdder(a=A[2],b=B[2],c=c1,sum=s2,carry=c2);
FullAdder(a=A[3],b=B[3],c=c2,sum=s3,carry=c3);
FullAdder(a=A[4],b=B[4],c=c3,sum=s4,carry=c4);
FullAdder(a=A[5],b=B[5],c=c4,sum=s5,carry=c5);
FullAdder(a=A[6],b=B[6],c=c5,sum=s6,carry=c6);
FullAdder(a=A[7],b=B[7],c=c6,sum=s7,carry=c7);
FullAdder(a=A[8],b=B[8],c=c7,sum=s8,carry=c8);
FullAdder(a=A[9],b=B[9],c=c8,sum=s9,carry=c9);
FullAdder(a=A[10],b=B[10],c=c9,sum=s10,carry=c10);
FullAdder(a=A[11],b=B[11],c=c10,sum=s11,carry=c11);
FullAdder(a=A[12],b=B[12],c=c11,sum=s12,carry=c12);
FullAdder(a=A[13],b=B[13],c=c12,sum=s13,carry=c13);
FullAdder(a=A[14],b=B[14],c=c13,sum=s14,carry=c14);
FullAdder(a=A[15],b=B[15],c=c14,sum=s15,carry=c15);
//Adding the two 16 bit number assuming the input carry is 1
FullAdder(a=A[0],b=B[0],c=true,sum=s16,carry=c16);
FullAdder(a=A[1],b=B[1],c=c16,sum=s17,carry=c17);
FullAdder(a=A[2],b=B[2],c=c17,sum=s18,carry=c18);
FullAdder(a=A[3],b=B[3],c=c18,sum=s19,carry=c19);
FullAdder(a=A[4],b=B[4],c=c19,sum=s20,carry=c20);
FullAdder(a=A[5],b=B[5],c=c20,sum=s21,carry=c21);
```

```
FullAdder(a=A[6],b=B[6],c=c21,sum=s22,carry=c22);
FullAdder(a=A[7],b=B[7],c=c22,sum=s23,carry=c23);
FullAdder(a=A[8],b=B[8],c=c23,sum=s24,carry=c24);
FullAdder(a=A[9],b=B[9],c=c24,sum=s25,carry=c25);
FullAdder(a=A[10],b=B[10],c=c25,sum=s26,carry=c26);
FullAdder(a=A[11],b=B[11],c=c26,sum=s27,carry=c27);
FullAdder(a=A[12],b=B[12],c=c27,sum=s28,carry=c28);
FullAdder(a=A[13],b=B[13],c=c28,sum=s29,carry=c29);
FullAdder(a=A[14],b=B[14],c=c29,sum=s30,carry=c30);
FullAdder(a=A[15],b=B[15],c=c30,sum=s31,carry=c31);
//Selecting the output carry based on the input carry
Mux(a=c15,b=c31,sel=Cin,out=Cout);
//Selecting the output sum based on the input carry
Mux(a=s0,b=s16,sel=Cin,out=S[0]);
Mux(a=s1,b=s17,sel=Cin,out=S[1]);
Mux(a=s2,b=s18,sel=Cin,out=S[2]);
Mux(a=s3,b=s19,sel=Cin,out=S[3]);
Mux(a=s4,b=s20,sel=Cin,out=S[4]);
Mux(a=s5,b=s21,sel=Cin,out=S[5]);
Mux(a=s6,b=s22,sel=Cin,out=S[6]);
Mux(a=s7,b=s23,sel=Cin,out=S[7]);
Mux(a=s8,b=s24,sel=Cin,out=S[8]);
Mux(a=s9,b=s25,sel=Cin,out=S[9]);
Mux(a=s10,b=s26,sel=Cin,out=S[10]);
Mux(a=s11,b=s27,sel=Cin,out=S[11]);
Mux(a=s12,b=s28,sel=Cin,out=S[12]);
Mux(a=s13,b=s29,sel=Cin,out=S[13]);
Mux(a=s14,b=s30,sel=Cin,out=S[14]);
Mux(a=s15,b=s31,sel=Cin,out=S[15];
```