FUNDAMENDTAL OF DIGITAL SYSTEM FINAL PROJECT REPORT
DEPARTMENT OF ELECTRICAL ENGINEERING
UNIVERSITAS INDONESIA

PIPELINE ALU WITH MULTI LEVEL ARITHMETIC OPTIMIZATION

GROUP 17

HAITSAM AHMAD FAKHRI	**2406357740**
SAKABUDI MUHAMMAD	**2406429683**
SABBIA MEILANDRI PUTRI D.	**2406351131**
NOVATAMA EKA FITRIA	**2406346466**

KATA PENGANTAR

Puji syukur kami panjatkan kepada Tuhan Yang Maha Esa karena atas rahmat dan karunia-Nya, kami dapat menyelesaikan proyek akhir mata kuliah Praktikum Perancangan Sistem Digital. Proyek akhir ini berjudul “Pipeline ALU with Multi-Level Arithmetic Optimization”. Penyusunan proyek ini bertujuan untuk menerapkan konsep-konsep perancangan sistem digital menggunakan bahasa VHDL sesuai modul pembelajaran yang meliputi perancangan menggunakan dataflow style, behavioral style, structural design, testbench, looping construct, procedure dan function, finite state machine (FSM), serta pendekatan microprogramming.

Pemilihan topik Pipeline ALU didasarkan pada kebutuhan untuk mengembangkan suatu rangkaian komputasi yang lebih efisien dan terstruktur, serta mempertimbangkan kesesuaian dengan karakteristik FPGA sebagai platform implementasi logika digital. Pipeline ALU dirancang sebagai hardware accelerator yang mampu melakukan operasi aritmatika dasar secara paralel melalui beberapa tahapan sehingga dapat meningkatkan throughput perhitungan.

Dalam proses penyusunan proyek ini, kami banyak memperoleh pemahaman mendalam mengenai perancangan rangkaian aritmatika, pemodelan RTL, manajemen data antar-stage pipeline, serta pengujian melalui testbench untuk memastikan validitas desain. Penulis mengucapkan terima kasih kepada asisten laboratorium dan seluruh pihak yang telah memberikan bimbingan serta dukungan selama pengerjaan proyek ini. Semoga laporan ini dapat bermanfaat dan menambah pengetahuan dalam bidang perancangan sistem digital berbasis FPGA.

Depok, 11 December 2025

Kelompok 17

DAFTAR ISI

BAGIAN 1

PENDAHULUAN

1.1 LATAR BELAKANG

Proyek ini berangkat dari kenyataan bahwa sistem digital modern bergerak semakin cepat namun tetap dituntut untuk efisien. Banyak perangkat yang kita gunakan tiap hari seperti ponsel, laptop, dan bahkan perangkat kecil seperti smartwatch bergantung pada kemampuan prosesor untuk memproses banyak instruksi dalam waktu yang sangat singkat. Di balik semua itu ada konsep pemrosesan terstruktur yang membuat kinerja tetap stabil meskipun beban kerja terus bertambah. Ide inilah yang akhirnya mendorong kami untuk mengeksplor cara kerja unit komputasi secara lebih mendalam.

Saat mempelajari cara prosesor bekerja, kami menemukan bahwa metode eksekusi instruksi tradisional sering kali menghabiskan banyak waktu karena satu instruksi harus selesai sepenuhnya sebelum instruksi berikutnya boleh mulai masuk. Jika dianalogikan, prosesnya mirip seperti antrean panjang yang hanya melayani satu orang sampai tuntas baru memanggil pelanggan berikutnya. Pola ini jelas kurang cocok untuk kebutuhan sistem saat ini yang menuntut respons hampir instan. Ketertarikan pada masalah ini membawa kami pada konsep pipeline yang ternyata sudah menjadi kunci peningkatan performa pada banyak arsitektur modern.

Gagasan pipeline menarik karena menawarkan solusi sederhana namun sangat efektif. Dengan memecah satu tugas kompleks menjadi beberapa tahap yang dapat berjalan bersamaan, sebuah unit komputasi bisa menghasilkan output di setiap siklus clock setelah pipeline terisi. Konsep ini membuat pemrosesan instruksi terasa lebih halus dan tidak lagi berhenti hanya karena satu instruksi membutuhkan tahap tertentu yang lebih lama. Dari sinilah muncul keinginan untuk memahami bagaimana sebuah pipeline dapat bekerja di ALU yang selama ini sering dianggap komponen yang hanya melakukan operasi dasar.

Dalam proses pembelajaran kami menemukan bahwa ALU tidak selalu sesederhana yang dibayangkan. Meskipun tugas utamanya melakukan operasi aritmatika dan logika, pemrosesannya bisa sangat padat ketika banyak instruksi harus diproses berturut turut. Ketika ALU tidak dioptimalkan, bottleneck dapat muncul dan membuat keseluruhan sistem menjadi

lambat. Tantangan inilah yang membuat kami tertarik membuat versi pipeline yang lebih terstruktur agar eksekusinya bisa berlangsung lebih efisien.

Penggunaan empat tahap pipeline yaitu Decode, Execute, Flag Generation, dan Write Back dirasa paling cocok untuk skenario ini. Pembagian ini memberikan aliran kerja yang jelas tanpa membuat desain menjadi rumit. Setiap tahap memiliki tanggung jawab spesifik sehingga instruksi dapat mengalir terus menerus selama clock berjalan. Pendekatan ini juga membantu mengurangi beban kerja pada satu blok sekaligus menjaga ritme pemrosesan tetap stabil.

Kami melihat bahwa konsep ini tidak hanya relevan untuk tugas akademik tetapi juga mencerminkan kebutuhan dunia nyata. Banyak aplikasi modern seperti pemrosesan data real time, game engine, ataupun sistem embedded pada perangkat pintar membutuhkan pemrosesan instruksi tanpa jeda. Pipeline ALU yang kami rancang berusaha meniru mekanisme efisiensi yang diterapkan di sistem tersebut sehingga proyek ini memiliki nilai praktis sekaligus edukatif.

Selain itu, proyek ini memberi kesempatan bagi kami untuk memahami bagaimana sinyal serta data mengalir di dalam sistem digital. Sering kali teori hanya memberikan gambaran besar sementara implementasi pipeline benar benar memperlihatkan interaksi antartahap yang terjadi secara sinkron. Pengalaman melihat data masuk, diproses, dan keluar secara bertahap memberikan pemahaman yang jauh lebih konkret tentang arsitektur prosesor.

Kami juga menyadari bahwa optimisasi pada level ALU bisa membuka banyak kemungkinan lanjutan. Dengan pipeline yang bekerja baik, instruksi dapat diproses secara berurutan tanpa saling menghambat sehingga performa keseluruhan sistem menjadi lebih baik. Dari sini kami mulai memahami mengapa pipeline menjadi fondasi penting pada mikroprosesor masa kini dan semakin tertarik untuk mengembangkan desain kami agar lebih mendekati implementasi nyata.

Penerapan register antartahap menjadi bagian yang tidak kalah menarik karena inilah kunci utama agar instruksi tetap aman selama berpindah dari satu tahap ke tahap berikutnya. Register pipeline berfungsi seperti checkpoint yang memastikan setiap data tertata rapi sebelum melanjutkan proses. Pengalaman merancang bagian ini membuat kami lebih peka terhadap detail kecil dalam desain digital yang sering kali menentukan stabil atau tidaknya suatu sistem.

Melalui proyek ini kami ingin menunjukkan bahwa optimisasi kecil pada struktur ALU dapat membawa dampak besar terhadap efisiensi pemrosesan. Pipeline menjadikan aliran instruksi lebih teratur dan meminimalkan waktu tunggu sehingga sistem dapat menghasilkan output secara konsisten. Dengan memahami prinsip dasarnya kami berharap dapat mengembangkan kemampuan desain sistem digital yang lebih kompleks di masa depan serta melihat hubungan nyata antara teori arsitektur komputer dan performa perangkat yang kita gunakan.

1.2 RUMUSAN MASALAH

Berdasarkan latar belakang yang telah dijelaskan, maka rumusan masalah dalam perancangan proyek akhir ini adalah sebagai berikut:

1. Bagaimana merancang ALU yang bisa menjalankan beberapa instruksi secara berurutan tanpa harus menunggu instruksi sebelumnya selesai?
2. Bagaimana membagi proses kerja ALU menjadi beberapa tahap yang jelas supaya eksekusi instruksi bisa lebih cepat?
3. Bagaimana memastikan setiap tahap pipeline tetap sinkron dan tidak saling mengganggu saat banyak instruksi berjalan bersamaan?
4. Bagaimana membaca operand dan menjalankan operasi aritmatika dengan benar di dalam struktur pipeline?
5. Bagaimana menjaga keakuratan hasil dan flag saat hasil operasi melewati beberapa tahap hingga akhirnya ditulis kembali ke register?

1.3 TUJUAN

Tujuan dari proyek ini adalah sebagai berikut:

1. Mengimplementasikan 4-Stage ALU menggunakan VHDL.
2. Mengimplementasikan konsep pipelining untuk memastikan instruksi bisa selesai.
3. Mencoba fungsi sistem dengan Vivado.

1.4 PEMBAGIAN TUGAS

Peran dan tanggung jawab yang diberikan kepada anggota kelompok adalah sebagai berikut:

Roles	Responsibilities	Person
Hazard Detection Unit & Flag Generation	Membuat logika untuk mendeteksi jika adanya tabrakan data antar instruksi. Menghitung status hasil dari operasi.	Haitsam Ahmad Fakhri
Execution & Pipeline	Membuat logika operasi matematika dan logika. Membuat penghubung antar bagian agar data bisa mengalir dari execution ke flag.	Sabbia Meilandri P.D
Decode & Memory	Menerjemahkan instruksi masuk untuk memisahkan opcode dan alamat register. Membuat tempat penyimpanan data.	Novatama Eka Fitria
Testbench	Membuat Testbench untuk mencoba input dan output pada proses.	Sakabudi Muhammad

BAGIAN 2

PEMBAHASAN

2.1 PERALATAN

Alat-alat yang akan digunakan dalam proyek ini adalah sebagai berikut:

- Vivado
- ModelSim
- Quartus
- VHDL
- GitHub
- Visual Studio Code

2.2 IMPLEMENTASI

Implementasi pipeline pada proyek ini dilakukan dengan membagi proses aritmetika menjadi tiga tahap utama, yaitu Instruction Decode & Operand Fetch, Operation Execution, dan Result Writeback. Pada tahap pertama, sistem menerima input berupa opcode dan dua operand, kemudian melakukan proses decoding untuk menentukan jenis operasi aritmetika yang diperlukan. Data tersebut disimpan dalam register pipeline sehingga dapat diteruskan secara sinkron ke tahap berikutnya. Pada tahap kedua, operasi aritmetika dieksekusi berdasarkan opcode yang telah terdecode. Eksekusi dilakukan oleh modul ALU yang bersifat kombinational sehingga hasilnya dapat dihasilkan dalam satu siklus clock. Hasil operasi kemudian diteruskan ke register pipeline untuk diproses lebih lanjut. Tahap terakhir melakukan proses penyelesaian di mana hasil perhitungan disiapkan sebagai output dan disertai dengan sinyal valid sebagai penanda bahwa data tersebut sudah siap digunakan.

Cara kerja pipeline:

1. Instruction Decode & Operand Fetch

Cara kerja pipeline dimulai ketika sinyal in_valid aktif yang menandakan bahwa sebuah instruksi baru telah masuk ke sistem. Pada tahap ini, opcode dan operand ditangkap oleh stage pertama, kemudian dilakukan proses decoding untuk mengetahui operasi apa yang harus dijalankan. Data yang telah terdecode disimpan pada register pipeline agar dapat diteruskan ke tahap berikutnya.

2. Operation Execution

Di tahap ini, operasi aritmetika dieksekusi berdasarkan opcode yang telah didecode sebelumnya. Stage pertama pada saat yang sama sudah dapat kembali menerima instruksi baru sehingga dua instruksi dapat diproses secara paralel pada stage yang berbeda. Hasil eksekusi dari stage kedua kemudian disimpan kembali ke register pipeline.

3. Result Writeback

Pada siklus ketiga, instruksi pertama memasuki stage terakhir, yaitu tahap penulisan hasil. Pada tahap ini, data hasil operasi disiapkan sebagai output dan diberikan sinyal valid untuk menunjukkan bahwa hasil tersebut sudah sah dan siap digunakan. Pada saat yang bersamaan, stage pertama dan kedua kembali memproses instruksi-instruksi berikutnya sehingga pipeline tetap terisi secara berkelanjutan.

4. Pipeline Full & Throughput Maksimal

Setelah pipeline terisi penuh, sistem dapat menghasilkan satu hasil operasi setiap satu siklus clock, walaupun satu instruksi tetap membutuhkan tiga stage internal. Mekanisme ini meningkatkan throughput secara signifikan dan mengurangi waktu idle yang biasanya terjadi pada ALU sequential.

Modul yang digunakan:

a. Modul 2 Dataflow Style

Bagian EX pada kode menggunakan gaya dataflow karena operasi aritmetika dilakukan secara kombinational di dalam proses tanpa register internal. Operasi seperti ADD, SUB, AND, OR, dan XOR langsung dihitung berdasarkan sinyal input ex_opcode, ex_data1, dan ex_data2. Hasil operasi akan langsung tersedia pada sinyal ex_result_raw tanpa perlu menunggu siklus clock.

b. Modul 3 Behavioral Style

Seluruh register pipeline seperti ID ke EX, EX ke FG, dan FG ke WB dibuat menggunakan behavioral style berbasis process(clk). Gaya ini digunakan untuk mengelola perpindahan data antar-stage, penyimpanan register file, penanganan bubble, serta reset sistem. Selain itu, hazard detection juga ditangani dengan process behavioral yang memonitor dependensi antar register tujuan dan sumber.

c. Modul 4 Testbench

Testbench akan digunakan untuk menguji aliran instruksi melalui semua stage pipeline mulai dari ID, EX, FG, hingga WB. Testbench memberi input instr_in, instr_valid, serta pola register awal, lalu memverifikasi apakah output result_out dan flags_out sudah sesuai. Testbench juga berfungsi untuk memeriksa respons pipeline terhadap hazard dan bubble.

d. Modul 5 Structural Programming

Seluruh komponen pipeline seperti register file, hazard logic, ALU, dan pipeline register disatukan dalam satu arsitektur pipeline_alu. Struktur ini bersifat structural karena menghubungkan berbagai blok fungsional menjadi satu sistem pipeline. Modul ini yang mengatur urutan aliran data dari decode hingga write-back.

e. Modul 6 Looping Construct

Looping tidak terlalu dominan dalam kode ini, tetapi konsepnya digunakan secara implisit pada struktur array register file (reg_array). Selain itu, looping juga digunakan pada testbench (nantinya) untuk memberi stimulus instruksi berulang dan menguji kasus pipeline panjang.

f. Modul 7 Procedure & Function

Meskipun pada kode ini belum ditulis secara eksplisit, bagian ALU dan decoding dapat dengan mudah dipecah menjadi fungsi terpisah (misalnya function untuk operasi aritmetika atau untuk perhitungan flag). Penyusunan dengan function ini akan membuat kode lebih modular dan mudah diuji.

g. Modul 8 Finite State Machine (FSM)

FSM digunakan secara halus pada pipeline ini melalui kontrol valid, setiap stage memiliki sinyal *_valid yang menentukan kapan data harus diproses atau kapan bubble harus

disisipkan. Meskipun bukan FSM eksplisit dengan state diagram, perilaku valid/bubble ini secara fungsional menerapkan mekanisme state antar-stage.

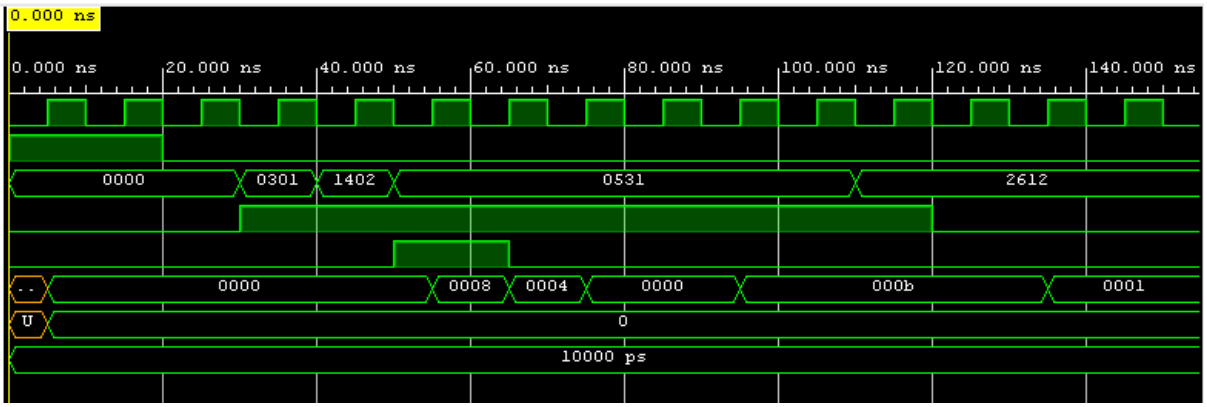
h. Modul 9 Microprogramming

Konsep microprogramming tidak digunakan penuh di kode ini, tetapi dapat diterapkan untuk memperluas operasi opcode. Misalnya, tabel micro-instruction dapat ditambahkan untuk operasi yang lebih kompleks. Pipeline ini kompatibel dengan pendekatan microprogramming apabila ingin dikembangkan.

BAGIAN 3

UJI COBA DAN ANALISIS

3.1 UJI COBA



Berdasarkan hasil simulasi pada waveform, kita bisa melihat prinsip kerja pipelining dimana instruksi dieksekusi secara overlapped. Setelah sinyal reset register file diinisialisasi dengan nilai awal (R0=5, R1=3, R2=1). Instruksi pertama ADD R3, R0, R1 (Opcode 0301) dieksekusi dan menghasilkan keluaran 0008 (8 desimal). Segera pada siklus clock berikutnya, hasil dari instruksi kedua SUB R4, R0, R2 (Opcode 1402) muncul dengan nilai 0004 (4 desimal). Munculnya hasil output pada setiap siklus clock membuktikan pipeline berfungsi secara efisien dalam kondisi tanpa konflik data mencapai throughput ideal satu instruksi per cycle.

Kita menguji hazard detection menggunakan ADD R5, R3, R1 (Opcode 0531). Instruksi ini memiliki ketergantungan data terhadap register R3 yang baru saja dimodifikasi oleh instruksi pertama. Seperti yang terlihat pada waveform hazard detection pada kondisi akan mengaktifkan sinyal stall_out menjadi logika '1' agar pipeline melakukan stalling selama beberapa cycle untuk menunggu data R3.

3.2 HASIL

Proyek ini mengimplementasikan 4-stage pipelined ALU yang menghasilkan tiga output utama yaitu result_out berupa hasil akhir operasi ALU 16-bit setelah melalui tahap Write Back, flags_out berupa 4-bit status flags yang terdiri dari Zero flag untuk menandai

hasil nol, Carry flag untuk operasi aritmatika (disederhanakan menjadi '0'), Overflow flag untuk deteksi overflow (disederhanakan menjadi '0'), dan Negative flag yang mengindikasikan bit MSB hasil, serta stall_out sebagai sinyal kontrol yang dikirim ke fetch unit ketika hazard terdeteksi untuk menghentikan sementara pengiriman instruksi baru. Dengan nilai inisialisasi register dimana R0=5, R1=3, dan R2=1, program menjalankan tiga instruksi pengujian: instruksi pertama ADD R3,R0,R1 (Opcode 0x0301) yang menjumlahkan R0 dan R1 menghasilkan R3=8, instruksi kedua SUB R4,R0,R2 (Opcode 0x1402) yang mengurangkan R2 dari R0 menghasilkan R4=4, dan instruksi ketiga ADD R5,R3,R1 (Opcode 0x0531) yang digunakan untuk menguji hazard detection karena membutuhkan R3 yang baru saja dihasilkan oleh instruksi pertama. Ketika instruksi ketiga mencoba membaca R3 sebelum instruksi pertama menyelesaikan tahap Write Back, sistem akan mendeteksi data hazard dan secara otomatis mengirim sinyal stall_out, menyisipkan bubble berupa instruksi NOP ke dalam pipeline hingga R3 selesai ditulis dengan nilai 0x0008 pada cycle kelima, baru kemudian instruksi ADD R5,R3,R1 dapat melanjutkan eksekusi dan menghasilkan R5=0x000B (11 desimal) dengan flags yang ter-update secara akurat. Setelah pipeline terisi penuh dan tidak ada hazard, sistem mampu menghasilkan satu hasil valid setiap satu cycle clock, menunjukkan peningkatan throughput yang signifikan dibandingkan eksekusi sekuensial.

Output yang dihasilkan oleh program ini bukan sekadar hasil eksekusi instruksi, melainkan bukti konkret bahwa arsitektur pipeline 4-stage mampu menangani kompleksitas pemrosesan instruksi modern dengan efisien dan akurat. Melalui analisis mendalam terhadap bagaimana setiap tahap pipeline menangani ketiga instruksi ini, kita dapat memahami secara komprehensif bagaimana program ini mengatasi tantangan fundamental dalam merancang ALU yang efisien, terstruktur, tersinkronisasi, akurat dalam operasi aritmatika, dan konsisten dalam menjaga integritas data hingga tahap akhir.

ALU ini menggunakan pipeline 4-stage (ID, EX, FG, WB) untuk memproses instruksi secara paralel. Setiap stage memiliki register pipeline sendiri, memungkinkan instruksi baru masuk saat instruksi sebelumnya dieksekusi. Setelah pipeline penuh (cycle ke-4), sistem menghasilkan 1 hasil per cycle, meningkatkan throughput dari 1 instruksi per 4 cycle menjadi 1 per cycle, atau 4x performa dibanding non-pipelined.

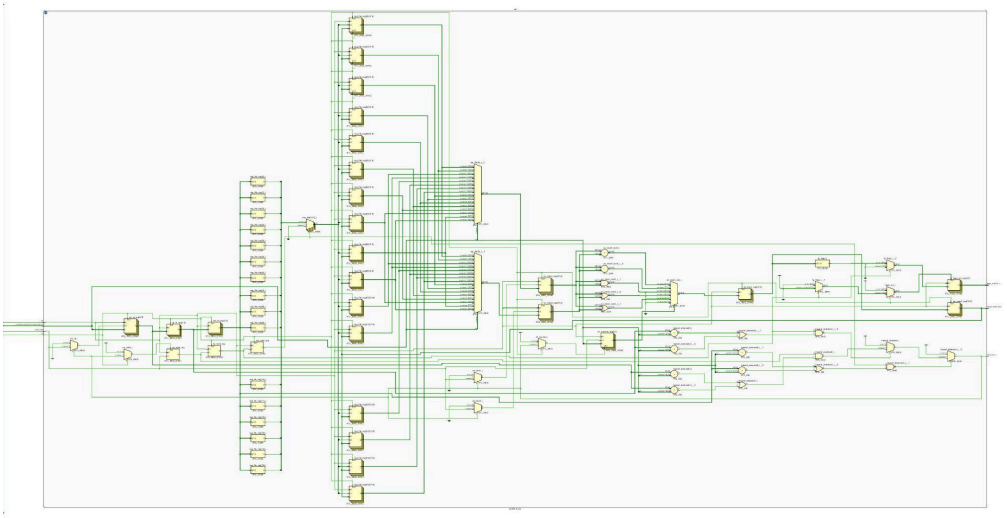
Proses dibagi menjadi 4 tahap seimbang, masing-masing selesai dalam 1 clock cycle:

- ID (Instruction Decode): Pecah instruksi 16-bit menjadi opcode, register tujuan/sumber, baca operand dari register file secara asynchronous.
- EX (Execute): Jalankan operasi aritmatika/logika dengan signed arithmetic 17-bit untuk menangkap carry/overflow.
- FG (Flag Generation): Hitung flags (Zero, Carry, Overflow, Negative) dari hasil final, bukan intermediate.
- WB (Write Back): Tulis hasil ke register file secara synchronous untuk hindari race condition.

Transfer data antar stage hanya pada rising edge clock dengan process terpisah, mencegah race condition. Hazard detection mendeteksi RAW (Read After Write) dan stall pipeline dengan NOP bubble (set ex_valid='0'), memastikan instruksi menunggu data valid. Valid signals (ex_valid, fg_valid, wb_valid) menandai data legitimate, menghindari pemrosesan bubble.

Register file (16 register, 16-bit) dibaca asynchronous di ID untuk kecepatan, ditulis synchronous di WB. Operasi di EX gunakan signed 17-bit untuk aritmatika (ADD, SUB) dan std_logic_vector untuk logika (AND, OR, XOR). Hazard detection pastikan operand terbaru dibaca, stall jika perlu, menjaga akurasi tanpa nilai stale.

Valid signals (ex_valid, fg_valid, wb_valid) propagasi bersama data, memastikan hanya operasi valid yang mempengaruhi register/flags. Bubble dari hazard tidak ubah state. Flags dihitung di FG dari hasil final 16-bit saat fg_valid='1', disimpan di register FG_WB, dan output synchronous dengan WB, menjamin konsistensi temporal dan akurasi meski ada bubble.



Schematic RTL yang dihasilkan dari sintesis kode VHDL pipeline ALU menunjukkan representasi visual dari arsitektur hardware yang akan diimplementasikan pada FPGA atau ASIC. Schematic ini memvisualisasikan seluruh komponen digital yang telah didefinisikan dalam kode, termasuk register file, pipeline registers, multiplexer untuk kontrol aliran data, unit aritmatika logika, dan logic hazard detection, yang semuanya terhubung melalui jalur data dan sinyal kontrol yang tersinkronisasi dengan clock.

1. Register File (reg_file_reg[0] hingga reg_file_reg[15])

   16 register synchronous 16-bit (R0-R15) dengan port clock (C), enable (CE), input (D), output (Q), dan kontrol reset/set. R0-R2 di-set awal (R0=0x0005, R1=0x0003, R2=0x0001), R3-R15 di-reset ke 0x0000. Dua multiplexer besar (reg_file[0]_i__0 dan reg_file[0]_i__1) untuk baca asynchronous paralel operand Rs1 dan Rs2, dengan selector dari alamat register yang di-decode.
2. Pipeline Registers - Stage ID ke EX

   Register synchronous menyimpan data decoded: ex_opcode_reg[3:0] (opcode 4-bit), ex_rd_reg[3:0] (alamat tujuan), ex_data1_reg[15:0] dan ex_data2_reg[15:0] (operand), serta ex_valid_reg (status valid). Multiplexer input (RTL_MUX) dengan jalur untuk bubble NOP (S=1'b1) saat hazard/reset, atau jalur normal dari ID, mengimplementasikan bubble insertion.
3. Execution Unit (ALU Operations)

   Serangkaian operator terhubung ke multiplexer ex_result_raw_i (6 input): RTL_ADD (penjumlahan 17-bit), RTL_SUB (pengurangan 17-bit), RTL_AND/OR/XOR (operasi bitwise 16-bit). Selector ex_opcode: 0000=ADD, 0001=SUB, 0010=AND, 0011=OR, 0100=XOR, default=NOP (output 0). Pemilihan hasil berdasarkan opcode di pipeline register EX.
4. Pipeline Registers - Stage EX ke FG

   Register menyimpan fg_result_reg[15:0] (hasil 16-bit dari operasi), fg_rd_reg[3:0] (alamat tujuan), dan fg_valid_reg (status valid). Komponen fg_flags_i (RTL_ROM) sebagai lookup table untuk flags 4-bit [Z, C, V, N], dengan multiplexer fg_flags_i__0/fg_flags_i__1: jalur valid (flags dari fg_result), jalur tidak valid (default B"0111" atau B"0000"). Flags hanya update untuk instruksi valid.
5. Pipeline Registers - Stage FG ke WB

   Register menyimpan wb_result_reg[15:0] (hasil final), wb_rd_reg[3:0] (alamat tujuan), wb_valid_reg (status valid), dan flags_out_reg[3:0] (flags untuk output).

Multiplexer flags_out_i: propagasi flags dari FG jika valid, atau reset default, memastikan konsistensi antara result yang ditulis dan flags output.

6. Hazard Detection Logic

   Serangkaian comparator (RTL_EQ) dan logic gates: 4 pasang comparator bandingkan id_rs1/id_rs2 dengan B"0001"/B"0010" untuk konflik spesifik, dikombinasikan dengan OR/AND untuk deteksi RAW di EX/FG/WB. Multiplexer hazard_detected_i/hazard_detected_i__0 gabungkan dengan ex_valid/fg_valid/wb_valid, hasilkan stall_out untuk hentikan fetch dan insert bubble NOP.

Data mulai dari instr_in[15:0] (instruksi 16-bit), dipecah via bit slicing menjadi opcode [15:12], rd [11:8], rs1 [7:4], rs2 [3:0]. Rs1/rs2 sebagai selector multiplexer baca register file ke id_data1/id_data2. Hazard detection paralel cek konflik rs1/rs2 dengan rd di EX/FG/WB. Jika tidak hazard, data decoded (opcode, rd, data1, data2) simpan ke ID_EX dengan ex_valid='1'; jika hazard, multiplexer pilih bubble (ex_valid='0'). Di EX, ex_opcode pilih hasil ALU (ADD/SUB/AND/OR/XOR) lalu, simpan ke fg_result_reg. Di FG, generate flags dari hasil simpan ke FG_WB. Di WB, jika wb_valid='1', tulis hasil ke register file via wb_rd, output flags via flags_out, selesai siklus.

Seluruh register synchronous dalam schematic terhubung ke sinyal clock (clk) yang sama, ditunjukkan dengan koneksi ke port C pada setiap RTL_REG_SYNC, memastikan semua transfer data antar stage terjadi secara bersamaan pada rising edge clock yang identik sehingga tidak ada race condition atau timing violation. Sinyal reset (rst) terdistribusi ke port RST dari semua pipeline registers dan register file, memungkinkan inisialisasi sistem ke kondisi awal yang terdefinisi dimana pipeline kosong (semua valid='0') dan register file terisi dengan nilai default. Sinyal chip enable (CE) pada register file dikontrol oleh wb_valid yang dipropagasi melalui multiplexer reg_file[15:0]_i, memastikan register file hanya diupdate ketika instruksi valid mencapai stage WB dan bukan ketika bubble NOP melewati pipeline. Sinyal instr_valid dari input eksternal dikombinasikan dengan hazard_detected melalui logic gates untuk menghasilkan kontrol yang menentukan apakah instruksi baru akan dimasukkan ke pipeline atau bubble akan diinsert, menciptakan mekanisme flow control yang dinamis namun tetap deterministik.

3.3 ANALISIS

Pada bagian paling luar desain, entitas pipeline_alu berfungsi sebagai wadah seluruh stage yang membentuk pipeline empat tahap. Port yang digunakan sudah menggambarkan alur kerja modul, mulai dari clk dan rst yang menandakan sistem sinkron, instr_in dan instr_valid sebagai pintu masuk instruksi, hingga result_out dan flags_out sebagai keluaran utama. Sinyal instr_valid menjaga supaya hanya instruksi yang benar benar siap yang masuk ke jalur pipeline sehingga tidak ada data acak yang ikut diproses. Sinyal stall_out berperan sebagai pesan ke blok luar bahwa pipeline sedang menahan instruksi baru karena ada ketergantungan data. Dengan struktur seperti ini, modul terlihat siap disambungkan ke unit fetch dan kontrol sederhana sehingga bisa menjadi bagian dari prosesor kecil yang operasional. Secara arsitektural, entitas ini sudah merangkum ide bahwa satu ALU bisa dipakai bergantian oleh banyak instruksi secara berlapis

Di dalam arsitektur, kumpulan constant opcode menjadi dasar bahasa mesin kecil yang dipahami oleh ALU. Nilai biner untuk operasi tambah, kurang, and, or, xor, serta satu kode khusus nop membuat setiap instruksi enam belas bit punya arti yang jelas. Adanya kode nop sangat penting karena dipakai sebagai gelembung yang berjalan di dalam pipeline ketika terjadi hazard. Dengan pendekatan ini, pipeline tidak diisi instruksi sembarangan saat stall, tetapi benar benar memasukkan operasi yang secara fungsional tidak mengubah isi register. Jumlah operasi memang belum banyak, namun sudah cukup untuk mewakili jalur aritmetika dan logika dasar yang sering dipakai dalam contoh prosesor sederhana. Penulisan opcode dalam bentuk constant juga memudahkan pengembangan, karena penambahan operasi baru cukup menambahkan satu constant dan satu cabang case di bagian ALU.

Register file disusun sebagai enam belas register umum dengan lebar enam belas bit, yang menjadi sumber dan tujuan utama data di dalam pipeline. Instruksi yang datang dipotong menjadi empat bagian, yaitu opcode, register tujuan, dan dua register sumber, sehingga format instruksi konsisten sepanjang desain. Dua sinyal id_data diisi dengan pembacaan langsung dari reg_file berdasarkan indeks rs1 dan rs2 yang diambil dari instruksi. Cara baca asinkron ini membuat operand selalu tersedia sebelum instruksi masuk ke tahap eksekusi, selama tidak ada hazard. Susunan seperti ini mengikuti pola arsitektur register register yang umum dipakai di prosesor sederhana, sehingga setiap instruksi berperilaku seperti operasi antara dua register dengan satu register tujuan. Secara visual, hal ini

memudahkan pembacaan waveform karena kita bisa mengaitkan setiap perubahan hasil dengan isi register tertentu.

Tahap decode sendiri ditulis sebagai logika kombinasi murni tanpa proses clock, sehingga setiap perubahan instruksi langsung memantul ke id_opcode dan operand di siklus yang sama. Kondisi ini membuat tahap pertama pipeline terasa ringan, karena tugasnya hanya memetakan bit dan membaca register. Hal ini sejalan dengan konsep bahwa decode pada prosesor sederhana biasanya tidak membutuhkan beberapa siklus, cukup satu siklus saja sebelum data disimpan di register pipeline. Keuntungan lain, penambahan variasi instruksi tidak mengganggu alur decode selama format dasar empat bit opcode dan tiga field register tetap dipertahankan. Dalam konteks praktis, pemisahan decode dan register file seperti ini memperjelas alur data dan memudahkan debug ketika ada nilai yang tidak sesuai ekspektasi.

Bagian deteksi hazard menjadi kunci supaya pipeline tidak menghasilkan perhitungan yang salah ketika ada ketergantungan antar instruksi. Blok proses yang mengecek rs1 dan rs2 terhadap rd di stage eksekusi, flag, dan write back secara langsung mengimplementasikan deteksi hazard baca setelah tulis. Setiap kali ada instruksi di tahap decode yang ingin membaca register yang sedang dihitung di salah satu dari tiga stage depan, sinyal hazard_detected dinaikkan. Logika ini memang cenderung konservatif karena bahkan konflik dengan stage write back juga ikut distall, namun pendekatan ini aman untuk tahap awal perancangan. Dengan cara ini, pipeline dijamin tidak akan memakai nilai lama di register ketika seharusnya sudah memakai hasil terbaru yang masih berada di jalur perhitungan.

Sinyal hazard_detected kemudian dipakai untuk membentuk stall_out yang dikirim ke luar, sekaligus mengontrol perilaku register pipeline ID ke EX. Ketika hazard terdeteksi, register pipeline tidak meneruskan instruksi yang sedang di decode, melainkan memasukkan instruksi nop dan menandai ex_valid nol. Dari sudut pandang pipeline, kondisi ini artinya ada gelembung yang sengaja disisipkan di jalur eksekusi sementara instruksi yang menimbulkan hazard tetap ditahan di tahap decode. Waveform simulasi memperlihatkan efeknya dalam bentuk bagian instruksi yang seolah diam untuk beberapa siklus dan area hasil yang menampilkan satu nilai nol di antara dua hasil operasi yang sah. Pola ini menjadi bukti bahwa unit deteksi hazard benar benar mengendalikan aliran instruksi dan memberi waktu bagi hasil sebelumnya untuk kembali ke register file.

Register pipeline dari ID ke EX menjadi jembatan yang memisahkan logika kombinasi decode dengan perhitungan di ALU. Di sini ada tiga kondisi utama yang diatur

dengan rapi, yaitu reset, hazard, dan instruksi valid. Saat reset, opcode diisi dengan nop dan semua sinyal data dikosongkan sehingga pipeline mulai dari keadaan bersih. Saat hazard, register pipeline memasukkan opcode nop dengan ex_valid nol sehingga stage eksekusi melihat slot kosong yang tidak perlu diproses lebih lanjut. Saat tidak ada hazard dan instruksi valid, barulah opcode dan dua operand disalin ke ex_opcode dan ex_data sehingga siap dihitung. Dengan pola ini, mudah untuk membedakan di waveform mana siklus yang memuat instruksi sungguhan dan mana siklus yang hanya berisi gelembung.

Tahap eksekusi berisi ALU yang mengerjakan operasi aritmetika dan logika berdasarkan ex_opcode dan dua operand. Sebelum dihitung, data diubah ke tipe signed dengan lebar tujuh belas bit sehingga operasi tambah dan kurang punya satu bit ekstra di bagian atas untuk menampung carry dan tanda. Hasil operasi disimpan di ex_result_raw yang juga memiliki lebar tujuh belas bit, sementara untuk operasi logika hanya enam belas bit bawah yang digunakan. Desain ini membuat satu bus hasil bisa dipakai oleh seluruh jenis operasi tanpa harus memecah menjadi beberapa sinyal berbeda. Meskipun implementasi carry dan overflow belum dieksploitasi penuh di stage flag, struktur ini sudah mengantisipasi kebutuhan tersebut jika proyek ingin dikembangkan.

Jika dikaitkan dengan nilai awal register, perilaku ALU sesuai dengan perhitungan manual. Saat reset, tiga register pertama berisi lima, tiga, dan satu yang kemudian dipakai sebagai operand. Instruksi pertama menjumlahkan register nol dan satu sehingga diharapkan menghasilkan delapan di register tiga, dan nilai ini memang muncul di waveform sebagai nol nol nol delapan. Instruksi kedua mengurangkan register nol dengan register dua dan menghasilkan empat di register empat, yang juga terlihat sebagai nol nol nol empat. Instruksi ketiga menjumlahkan register tiga yang baru saja dihitung dengan register satu sehingga hasilnya sebelas, dan di waveform muncul sebagai nilai heksadesimal nol nol nol b. Instruksi keempat melakukan operasi and antara tiga dan satu sehingga hasil satu tertulis ke register enam. Konsistensi antara teori dan simulasi ini menunjukkan bahwa ALU bekerja dengan benar untuk urutan instruksi yang diuji.

Register pipeline dari EX ke tahap flag membuat perhitungan aritmetika benar benar dipisahkan dari evaluasi flag. Hasil raw dari ALU diambil enam belas bit bawahnya lalu disimpan di fg_result, sementara nomor register tujuan disalin ke fg_rd, dan sinyal valid diteruskan. Pemisahan ini selaras dengan ide multi level arithmetic optimization karena setiap stage hanya memegang beban komputasi yang wajar sehingga frekuensi clock bisa dijaga

tanpa satu stage menjadi titik lemah. Waveform menunjukkan bahwa perpindahan nilai dari ex_result_raw ke fg_result terjadi secara bersih pada tepi naik clock, tanpa glitch yang mengganggu. Hal ini menandakan bahwa register pipeline sudah ditempatkan pada batas logika yang tepat.

Tahap flag generation memfokuskan diri pada interpretasi hasil yang sudah stabil di fg_result. Ketika fg_valid satu, modul ini mengecek apakah hasil sama dengan nol untuk menyalakan flag zero dan melihat bit paling tinggi untuk menentukan flag negatif. Dua flag lain yaitu carry dan overflow sementara diset nol sebagai bentuk penyederhanaan, namun empat bit flag sudah disiapkan sehingga di masa depan bit ke tujuh belas dari hasil ALU bisa dipakai untuk menjalankan logika yang lebih detail. Ketika tidak ada instruksi valid, seluruh flag dikosongkan sehingga tidak ada nilai acak yang mengalir ke port keluaran. Walaupun sederhana, tahap ini sudah cukup untuk menandai kondisi hasil nol atau bernilai negatif yang sering dipakai pada cabang kondisi di arsitektur prosesor.

Register pipeline dari tahap flag ke tahap tulis balik menyatukan semua informasi yang dibutuhkan sebelum menyentuh register file dan port luar. Pada setiap tepi naik clock, sinyal valid, nomor register tujuan, dan hasil perhitungan disalin ke wb_valid, wb_rd, dan wb_result. Di saat yang sama, flags_out diisi dengan nilai flag yang sudah dihitung sehingga lingkungan luar dapat memantau kondisi hasil tanpa perlu mengintip isi internal pipeline. Saat reset, seluruh sinyal ini diinisialisasi ke nol sehingga gelombang awal simulasi bersih dari hasil palsu. Tahap ini pada dasarnya adalah pintu keluar pipeline yang mengubah nilai sementara di jalur menjadi state arsitektural yang benar benar tersimpan.

Bagian tulis balik ke register file menyempurnakan alur data dengan menyalurkan hasil ke bank register. Di sini logika memanfaatkan sinyal wb_valid sebagai izin tulis sehingga hanya hasil instruksi yang benar benar selesai yang disimpan. Pada saat reset, register nol sampai dua diberi nilai lima, tiga, dan satu, sedangkan register tiga sampai lima belas diisi nol untuk menjaga konsistensi awal simulasi. Setelah itu, setiap instruksi yang mencapai tahap write back akan menyimpan hasilnya ke indeks register yang sesuai dengan wb_rd. Melalui mekanisme ini, jejak urutan instruksi bisa diikuti dengan cara membaca kembali isi register pada akhir simulasi, dan unit deteksi hazard pun memiliki dasar yang kuat untuk membandingkan register yang sedang dituju oleh instruksi yang masih berada di jalur.

Testbench yang digunakan menyusun skenario instruksi yang sengaja dirancang untuk menguji fungsi ALU sekaligus logika hazard. Setelah clock berjalan dan reset dilepas, testbench mengaktifkan instr_valid lalu mengirim instruksi pertama yang menjumlahkan register nol dan satu menuju register tiga. Dilanjutkan dengan instruksi kedua yang mengurangkan register nol dan dua menuju register empat sehingga pipeline mulai terisi. Instruksi ketiga dibuat bergantung pada hasil pertama karena memakai register tiga sebagai salah satu operand, padahal pada saat itu hasilnya belum sempat masuk kembali ke register file. Setelah instruksi ketiga dikirim, testbench menjalankan loop yang berkali kali mengecek nilai stall_out sambil tetap mempertahankan instruksi yang sama di instr_in ketika stall masih aktif. Baru setelah hazard dianggap selesai, instruksi keempat dikirim untuk melakukan operasi and antara register satu dan dua.

Waveform hasil simulasi merekam seluruh skenario ini secara visual. Pada jalur instruksi terlihat urutan nilai heksadesimal yang mewakili empat instruksi, dengan bagian yang tampak bertahan lebih lama saat instruksi ketiga tertahan karena hazard. Jalur hasil menunjukkan urutan nilai nol pada awal, lalu delapan, empat, nol kembali, sebelas, dan terakhir satu yang mencerminkan hasil penjumlahan, pengurangan, gelembung, penjumlahan bergantung, dan operasi and. Nilai nol yang muncul di tengah setelah empat merupakan tanda bahwa gelembung nop benar benar berjalan melewati pipeline saat stall berlangsung. Di bagian flag, nilai tetap terkontrol dan tidak muncul keadaan tak terdefinisi setelah reset, yang berarti tahap flag dan penyalurannya ke luar sudah bekerja dengan stabil. Secara umum, kombinasi desain kode dan hasil simulasi menunjukkan bahwa pipeline empat tahap dengan deteksi hazard berbasis stall dan gelembung ini sudah berperilaku sesuai rancangan dan siap menjadi dasar untuk pengembangan fitur lanjutan seperti forwarding atau deteksi carry dan overflow yang lebih lengkap.

BAGIAN 4

KESIMPULAN

Proyek ini berhasil merancang dan mengimplementasikan Arithmetic Logic Unit (ALU) pipelined empat tahap yang inovatif, yang secara signifikan meningkatkan efisiensi pemrosesan instruksi dalam sistem digital. Dengan mengatasi keterbatasan eksekusi instruksi satu per satu yang lambat dan tidak efisien, desain ini memungkinkan beberapa instruksi diproses secara paralel tanpa harus menunggu instruksi sebelumnya selesai sepenuhnya. Hal ini dicapai melalui pembagian proses kerja ALU menjadi empat tahap terstruktur: Instruction Decode (ID), Execute (EX), Flag Generation (FG), dan Write Back (WB), masing-masing dirancang untuk diselesaikan dalam satu siklus clock. Setelah pipeline terisi penuh pada siklus keempat, sistem mampu menghasilkan satu hasil operasi setiap siklus clock, meningkatkan throughput hingga empat kali lipat dibandingkan ALU non-pipelined yang hanya memproses satu instruksi per empat siklus. Sinkronisasi antar tahap dijaga melalui register pipeline khusus untuk setiap stage, yang menyimpan data instruksi secara synchronous pada rising edge clock, serta sinyal kontrol seperti clock (clk), reset (rst), dan valid signals (ex_valid, fg_valid, wb_valid) untuk mencegah race condition atau data corruption. Mekanisme deteksi hazard Read After Write (RAW) secara otomatis mendeteksi konflik register sumber dengan instruksi sebelumnya di stage EX, FG, atau WB, lalu menyisipkan bubble NOP (dengan ex_valid='0') untuk stall pipeline sementara, memastikan integritas data tanpa mengganggu aliran paralel.

Pembacaan operand dilakukan secara asynchronous di stage ID melalui register file 16-bit (dengan 16 register R0-R15), memungkinkan akses cepat tanpa delay clock, sementara operasi aritmatika di stage EX menggunakan tipe data signed 17-bit untuk menangkap carry dan overflow pada operasi ADD/SUB, serta std_logic_vector 16-bit untuk operasi logika seperti AND, OR, dan XOR. Flag status (Zero, Carry, Overflow, Negative) dihitung secara terpisah di stage FG berdasarkan hasil final 16-bit, bukan nilai intermediate, dan dipropagasi bersama hasil hingga writeback ke register file secara synchronous. Schematic RTL menunjukkan komponen utama seperti multiplexer untuk kontrol aliran data, operator ALU (RTL_ADD, RTL_SUB, dll.), dan logic gates untuk hazard detection, semuanya terhubung dengan sinyal instr_valid dan stall_out untuk flow control dinamis. Register file diinisialisasi dengan nilai default (R0=0x0005, R1=0x0003, R2=0x0001, lainnya 0x0000) dan hanya

diupdate saat wb_valid='1', menghindari penulisan bubble. Rumusan masalah proyek mencakup tantangan seperti merancang ALU untuk eksekusi instruksi berurutan tanpa penundaan, membagi proses ke tahap jelas untuk kecepatan, memastikan sinkronisasi tanpa gangguan, membaca operand dan menjalankan operasi aritmatika dengan benar, serta menjaga akurasi hasil dan flag melalui pipeline.

REFERENSI

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann, 2003. [Online]. Available: https://books.google.co.id/books?id=R7Frpn3g9AEC

[2] S. Brown and Z. Vranesic, *Fundamentals of Digital Logic with VHDL Design*, 3rd ed. New York, NY, USA: McGraw–Hill, 2013. [Online]. Available: https://www.mheducation.com/highered/product/fundamentals-digital-logic-vhdl-design-brown-vranesic/M9780073380544.html

[3] P. J. Ashenden, *The Designer's Guide to VHDL*, 3rd ed. Burlington, MA, USA: Morgan Kaufmann, 2008. [Online]. Available: https://books.google.co.id/books?id=X9CkF5M0VfIC

[4] AMD Xilinx, "Vivado Design Suite User Guide Synthesis (UG901)," 2023. [Online]. Available: https://docs.amd.com/r/en-US/ug901-vivado-synthesis

[5] AMD Xilinx, "Vivado Design Suite User Guide Logic Simulation (UG900)," 2023. [Online]. Available: https://docs.amd.com/r/en-US/ug900-vivado-logic-simulation

[6] Intel Corporation, "Quartus Prime Standard Edition Handbook Volume 1: Design and Synthesis," 2017. [Online]. Available: https://www.cs.cornell.edu/courses/cs3410/2019sp/files/quartus-handbook-vol-1.pdf

[7] Siemens EDA, "ModelSim Simulation User's Manual," 2022. [Online]. Available: https://www.sw.siemens.com/en-US/media/modelsim-simulation-user-manual_tcm195-105730.pdf

[8] Tutorialspoint, "Digital Circuits – Finite State Machines," 2020. [Online]. Available: https://www.tutorialspoint.com/digital_circuits/digital_circuits_finite_state_machines.htm

[9] D. S. Lång, "Finite-State Machines," KTH Royal Institute of Technology, lecture notes, 2011. [Online]. Available: https://www.csc.kth.se/utbildning/kth/kurser/DD1361/ak11/forelasningar/fsm.pdf

[10] A. S. Susin, “EEL7120 – VHDL I Lecture 8: VHDL Test Benches,” Universidade Federal de Santa Catarina, 2014. [Online]. Available: https://www.inf.ufsc.br/~susin/files/digital2/08-VHDL-Testbenches.pdf

[11] I. Koren, “Microprogrammed Control,” Dept. of Electrical & Computer Engineering, Univ. of Massachusetts Amherst, 2006. [Online]. Available: https://euler.ecs.umass.edu/ece354/pdf/MicroprogrammedControl.pdf

[12] GeeksforGeeks, “Computer Organization – Arithmetic Logic Unit (ALU),” 2019. [Online]. Available: https://www.geeksforgeeks.org/computer-organization-arithmetic-logic-unit-alu/

[13] GeeksforGeeks, “Computer Organization – Data Hazards and Its Handling Methods,” 2019. [Online]. Available: https://www.geeksforgeeks.org/computer-organization-data-hazards-and-its-handling-methods/

[14] CS 255 Computer Architecture, Emory University, “Pipelining 2,” lecture slides, 2017. [Online]. Available: https://www.cs.emory.edu/~cheung/Courses/355/Syllabus/9-pipe/p2-pipeline.pdf

[15] S. Bogdan and S. Gacovski, “An Analysis of the Efficiency of Pipelined Processing Model in the Design of Integrated Devices,” *Engineering, Technology & Applied Science Research*, vol. 11, no. 4, pp. 7385–7390, 2021. [Online]. Available: https://www.etasr.com/index.php/ETASR/article/view/4121

[16] Emory University, “The Pipelined CPU Approach,” CS 355 – Computer Architecture, online lecture notes. [Online]. Available: https://www.cs.emory.edu/~cheung/Courses/355/Syllabus/7-pipeline/pipeline.html

[17] Emory University, “Speedup Achieved through Pipelining,” CS 355 – Computer Architecture, online lecture notes. [Online]. Available: https://www.cs.emory.edu/~cheung/Courses/355/Syllabus/7-pipeline/speedup.html

[18] Arvind and K. Asanović, “Pipeline Hazards,” *6.823 Computer System Architecture*, Massachusetts Institute of Technology, lecture 6, 2005. [Online]. Available:

https://ocw.mit.edu/courses/6-823-computer-system-architecture-fall-2005/16eb29d3b9c087566a1a28aca412bf02_l06_pipeline.pdf

[19] C. Kozyrakis, "Pipeline Hazards," *EE108b: Digital Systems Design*, Stanford University, lecture 9. [Online]. Available: https://web.stanford.edu/class/archive/ee/ee108b/ee108b.1082/handouts/lect.09.PipelinedHazards.pdf

[20] University of Washington, "Pipeline Hazards," *CSE 410: Computer Systems*, lecture 11. [Online]. Available: https://courses.cs.washington.edu/courses/cse410/22wi/lectures/11-pipelineHazards.pdf

[21] H. Weatherspoon, "Lec 9: Pipeline Hazards," *CS3410: Computer System Organization and Programming*, Cornell University, 2012. [Online]. Available: https://www.cs.cornell.edu/courses/cs3410/2012sp/lecture/10-hazards-w.pdf

[22] Z. Wang, "Lecture 9 – Pipeline Hazards," *CDA3101: Introduction to Computer Organization*, Florida State University, Fall 2017. [Online]. Available: https://www.cs.fsu.edu/~zwang/files/cda3101/Fall2017/Lecture9_cda3101.pdf

[23] M. D. Sinclair, "CS/ECE 552: Pipeline Hazards," *Computer Architecture*, University of Wisconsin–Madison, lecture notes. [Online]. Available: https://pages.cs.wisc.edu/~sinclair/courses/cs552/lectVideos/cs552-lec16c-pipelineHazardCtrl/cs552-16-pipeline_hazards.pdf

[24] Emory University, "Executing an ALU Instruction: Part 2 – a Register and a Constant," CS 355 – Computer Architecture, online lecture notes. [Online]. Available: https://www.cs.emory.edu/~cheung/Courses/355/Syllabus/7-pipeline/alu2.html

[25] R. E. Kessler, E. J. McLellan, and D. A. Webb, "The Alpha 21264 Microprocessor Architecture," Compaq Computer Corporation, technical paper. [Online]. Available: https://manx-docs.org/collections/antonio/dec/The_Alpha_21264_Microprocessor_Architecture.pdf

LAMPIRAN

Lampiran A: Project Schematic