TUGAS AKHIR - EC184801

SOFT MICROPROCESSOR CORE DENGAN *PIPELINE* LIMA TAHAP BERDASARKAN RISC-V *BASE INTEGER INSTRUCTION SET ARCHITECTURE* DALAM VHDL

Aaron Elson Phangestu
NRP 0721 18 4000 0041

Dosen Pembimbing
Dr. Ir. Totok Mujiono, M.IKom.
NIP 196504221989031001
Ahmad Zaini ST, M.T.
NIP 197504192002121003

Program Studi Teknik Komputer
Departemen Teknik Komputer
Fakultas Teknologi Elektro dan Informatika Cerdas
Institut Teknologi Sepuluh Nopember
Surabaya
2022

TUGAS AKHIR - EC184801

SOFT MICROPROCESSOR CORE DENGAN *PIPELINE* LIMA TAHAP BERDASARKAN RISC-V *BASE INTEGER INSTRUCTION SET ARCHITECTURE* DALAM VHDL

Aaron Elson Phangestu
NRP 0721 18 4000 0041

Dosen Pembimbing
Dr. Ir. Totok Mujiono, M.IKom.
NIP 196504221989031001
Ahmad Zaini ST, M.T.
NIP 197504192002121003

Program Studi Teknik Komputer
Departemen Teknik Komputer
Fakultas Teknologi Elektro dan Informatika Cerdas
Institut Teknologi Sepuluh Nopember
Surabaya
2022

[Halaman ini sengaja dikosongkan]

FINAL PROJECT - EC184801

FIVE-STAGE PIPELINED 32-BIT RISC-V BASE INTEGER INSTRUCTION SET ARCHITECTURE SOFT MICROPROCESSOR CORE IN VHDL

Aaron Elson Phangestu
NRP 0721 18 4000 0041

Advisor
Dr. Ir. Totok Mujiono, M.IKom.
NIP 196504221989031001
Ahmad Zaini ST, M.T.
NIP 197504192002121003

Study Program Computer Engineering
Department of Computer Engineering
Faculty of Intelligent Electrical and Informatics Technology
Sepuluh Nopember Institute of Technology
Surabaya

2022

[Halaman ini sengaja dikosongkan]

LEMBAR PENGESAHAN

SOFT MICROPROCESSOR CORE DENGAN *PIPELINE* LIMA TAHAP BERDASARKAN RISC-V *BASE INTEGER INSTRUCTION SET ARCHITECTURE* DALAM VHDL

TUGAS AKHIR

Diajukan untuk memenuhi salah satu syarat memperoleh gelar Sarjana Teknik pada Program Studi S-1 Teknik Komputer Departemen Teknik Komputer Fakultas Teknologi Elektro dan Informatika Cerdas Institut Teknologi Sepuluh Nopember

Oleh: Aaron Elson Phangestu
NRP. 0721 18 4000 0041

Disetujui oleh Tim Penguji Tugas Akhir:

Dr. Ir. Totok Mujiono, M.IKom. NIP: 19650422 198903 1 001	(Pembimbing I) .....
Ahmad Zaini, S.T., M.Sc. NIP: 19750419 200212 1 003	(Pembimbing II) .....
Mochamad Hariadi, ST., M.Sc., Ph.D. NIP: 19691209 199703 1 002	(Penguji I) .....
Atar Fuady Babgei, S.T., M.Sc. NIP: 19891111 201812 1 001	(Penguji II) .....
Dion Hayu Fandiantoro, S.T., M.T. NPP: 1994202011064	(Penguji III) .....

Mengetahui,
Kepala Departemen Teknik Komputer FTEIC - ITS

Dr. Supeno Mardi Susiki Nugroho, S.T., M.T.
NIP. 19700313 199512 1 001

SURABAYA
Juli, 2022

[Halaman ini sengaja dikosongkan]

APPROVAL SHEET

FIVE-STAGE PIPELINED 32-BIT RISC-V BASE INTEGER INSTRUCTION SET ARCHITECTURE SOFT MICROPROCESSOR CORE IN VHDL

FINAL PROJECT

Submitted to fullfill one of the requirements for obtaining Engineering degree at Undergraduate Study Program of Computer Engineering Department of Computer Engineering Faculty of Intelligent Electrical and Informatics Technology Sepuluh Nopember Institute of Technology

By: Aaron Elson Phangestu
NRP. 0721 18 4000 0041

Approved by Final Project Examiner Team:

Dr. Ir. Totok Mujiono, M.IKom. NIP: 19650422 198903 1 001	(Advisor I) .....
Ahmad Zaini, S.T., M.Sc. NIP: 19750419 200212 1 003	(Advisor II) .....
Mochamad Hariadi, ST., M.Sc., Ph.D. NIP: 19691209 199703 1 002	(Examiner I) .....
Atar Fuady Babgei, S.T., M.Sc. NIP: 19891111 201812 1 001	(Examiner II) .....
Dion Hayu Fandiantoro, S.T., M.T. NPP: 1994202011064	(Examiner III) .....

Acknowledged,
Head of Computer Engineering Department

Dr. Supeno Mardi Susiki Nugroho, S.T., M.T.
NIP. 19700313 199512 1 001

SURABAYA
July, 2022

[Halaman ini sengaja dikosongkan]

PERNYATAAN ORISINALITAS

Yang bertanda tangan dibawah ini:

Nama Mahasiswa / NRP	: Aaron Elson Phangestu / 0721 18 4000 0041
Departemen	: Teknik Komputer
Dosen Pembimbing	: Dr. Ir. Totok Mujiono, M.IKom.

Dengan ini menyatakan bahwa Tugas Akhir dengan judul "*Soft Microprocessor Core* dengan *Pipeline* Lima Tahap Berdasarkan RISC-V *Base Integer Instruction Set Architecture* dalam VHDL" adalah hasil karya sendiri, berfsifat orisinal, dan ditulis dengan mengikuti kaidah penulisan ilmiah.

Bilamana di kemudian hari ditemukan ketidaksesuaian dengan pernyataan ini, maka saya bersedia menerima sanksi sesuai dengan ketentuan yang berlaku di Institut Teknologi Sepuluh Nopember.

Surabaya, Juli 2022

Mengetahui Dosen Pembimbing	Mahasiswa
(Dr. Ir. Totok Mujiono, M.IKom) NIP. 196504221989031001	(Aaron Elson Phangestu) NRP. 0721 18 4000 0041

[Halaman ini sengaja dikosongkan]

STATEMENT OF ORIGINALITY

The undersigned below:

Name of student / NRP	: Aaron Elson Phangestu / 0721 18 4000 0041
Department	: Computer Engineering
Advisor	: Dr. Ir. Totok Mujiono, M.IKom.

Hereby declared that the Final Project with the title of "Five-Stage Pipelined 32-BIT RISC-V Base Integer Instruction Set Architecture Soft Microprocessor Core in VHDL" is the result of my own work, is original, and is written by following the rules of scientific writing.

If in future there is a discrepancy with this statement, then I am willing to accept sanctions in accordance with provisions that apply at Sepuluh Nopember Institute of Technology.

Surabaya, Juli 2022

Acknowledged Advisor	Student
(Dr. Ir. Totok Mujiono, M.IKom) NIP. 196504221989031001	(Aaron Elson Phangestu) NRP. 0721 18 4000 0041

[Halaman ini sengaja dikosongkan]

ABSTRAK

Nama Mahasiswa	:	Aaron Elson Phangestu
Judul Tugas Akhir	:	*Soft Microprocessor Core* dengan *Pipeline* Lima Tahap Berdasarkan RISC-V *Base Integer Instruction Set Architecture* dalam VHDL
Pembimbing	:	1. Dr. Ir. Totok Mujiono, M.IKom. 2. Ahmad Zaini, S.T, M.T.

Teknologi *proprietary* dengan perizinan yang rumit saat ini mendominasi industri mikroprosesor. Akibatnya, pengembang perangkat keras harus mencari alternatif sumber terbuka yang tersedia secara bebas. Dalam Tugas Akhir ini, penulis membahas implementasi inti prosesor lunak dengan *pipeline* lima tahap. *Processor core* menggunakan *Base Integer Instruction Set Architecture* RISC-V RV32I. RISC-V adalah ISA standar terbuka yang tersedia secara bebas untuk digunakan dan dimodifikasi. Untuk mengimplementasikan inti prosesor, penulis mengikuti metodologi desain FPGA. Pertama, penulis mengimplementasikan spesifikasi desain dengan Bahasa Deskripsi Perangkat Keras VHDL. Kemudian, penulis mensimulasikan desain di lingkungan simulasi ModelSim. Setelah verifikasi, penulis menganalisis penggunaan sumber daya, jalur kritis, dan frekuensi maksimum prosesor, lalu mengunggah inti prosesor ke Cyclone IV EP4CE6E22C FPGA yang sebenarnya. Inti CPU penulis berhasil menjalankan semua instruksi RV32I, kecuali instruksi FENCE, ECALL, dan CSR. Inti prosesor yang diusulkan berjalan pada 2115 LUT, 558 flip-flop, dan 67.608 bit memori, dengan frekuensi maksimum 62,95 MHz.

Kata Kunci: RISC-V, RV32I, *FPGA*, VHDL, *soft processor core*, *pipeline* lima tahap

[Halaman ini sengaja dikosongkan]

ABSTRACT

Name : Aaron Elson Phangestu
Title : *Five-Stage Pipelined 32-Bit RISC-V Base Integer Instruction Set Architecture Soft Microprocessor Core in VHDL*
Advisors : 1. Dr. Ir. Totok Mujiono, M.IKom.
2. Ahmad Zaini, S.T., M.T.

Proprietary technologies with complicated licensing currently dominate the microprocessor industry. As a result, we must seek out a freely available, open-source alternative. In this paper, we discussed the implementation of a five-stage pipelined soft processor core. The core uses the RISC-V RV32I Base Integer Instruction Set Architecture. RISC-V is an open standard ISA that is freely available to use and modify. To implement our processor core, we followed the FPGA design methodology. First, we implemented the design specification with the VHDL Hardware Description Language. Then, we simulated the design in the ModelSim simulation environment. Following the verification, we analyzed the resource usage, critical path, and maximum frequency of the processor, then uploaded the processor core to an actual Cyclone IV EP4CE6E22C FPGA. Our CPU core successfully executed all the RV32I instructions, except FENCE, ECALL, and CSR instructions. The proposed processor core runs on 2115 LUTs, 558 flip-flops, and 67,608 memory bits, with a maximum frequency of 62.95 MHz.

Keywords: RISC-V, RV32I, *FPGA*, VHDL, *soft processor core*, *five-stage pipeline*.

[Halaman ini sengaja dikosongkan]

KATA PENGANTAR

Puji dan syukur kehadirat Tuhan Yang Maha Esa atas segala karunia-Nya, penulis dapat menyelesaikan penelitian ini dengan judul ***Soft Microprocessor Core* dengan *Pipeline* Lima Tahap Berdasarkan RISC-V *Base Integer Instruction Set Architecture* dalam VHDL**.

Penelitian ini disusun dalam rangka pemenuhan syarat kelulusan mata kuliah Tugas Akhir di Departemen Teknik Komputer ITS, serta digunakan sebagai syarat menyelesaikan pendidikan Sarjana. Penelitian ini dapat diselesaikan tidak lepas dari bantuan berbagai pihak. Oleh karena itu, penulis mengucapkan terimakasih kepada:

1. Keluarga, Ibu, Bapak dan Saudara tercinta yang telah memberikan dorongan baik secara spiritual dan material dalam penyelesaian buku penelitian ini.
2. Bapak Dr. Supeno Mardi Susiki Nugroho, ST., MT. selaku Kepala Departemen Teknik Komputer, Fakultas Teknologi Elektro dan Informatika Cerdas, Institut Teknologi Sepuluh Nopember.
3. Bapak Dr. Ir. Totok Mujiono, M.IKom. selaku dosen pembimbing I dan Bapak Ahmad Zaini, S.T., M.T. selaku dosen pembimbing II yang selalu memberikan arahan selama mengerjakan penelitian tugas akhir ini.
4. Bapak-ibu dosen pengajar Departemen Teknik Komputer, atas pengajaran, bimbingan, serta perhatian yang diberikan kepada penulis selama ini.

Tidak ada manusia yang sempurna, untuk itu penulis memohon segenap kritik dan saran yang membangun. Semoga penelitian ini dapat memberikan manfaat bagi banyak orang.

Surabaya, Juli 2022

Aaron Elson Phangestu

[Halaman ini sengaja dikosongkan]

DAFTAR ISI

[Halaman ini sengaja dikosongkan]

DAFTAR GAMBAR

DAFTAR TABEL

BAB I
PENDAHULUAN

1.1 Latar Belakang

RISC-V merupakan salah satu arsitektur *open standard* yang digunakan untuk aplikasi akademis dan indsutri(Waterman et al., 2014). Arsitektur ini mulanya digunakan pada tahun 2010 di UC Berkeley untuk keperluan edukasi dan riset arsitektur komputer. Seiring waktu, RISC-V memperoleh traksi industri, dan pada tahun 2015 RISC-V Foundation hadir untuk mendorong adopsi RISC-V. Untuk meningkatkan adopsi, RISC-V telah didesain untuk mendukung *address space* 32-bit, 64-bit, dan 128-bit. RISC-V dapat dibagi menjadi beberapa *ISA base integer*, yang menyediakan target untuk *assembler*, *linker*, *compiler*, dan sistem operasi. RISC-V *Foundation* menyediakan *toolchain* yang mendukung fitur ini(Poorhosseini et al., 2020).

Pengguna dapat memanfaatkan desain RISC-V untuk berbagai keperluan, salah satunya menggunakannya suatu *soft processor* dengan mensintesisnya ke sebuah *FPGA*. *Field Programable Gate Array* (*FPGA*) merupakan suatu *integrated circuit* yang dapat dikonfigurasi oleh pengguna setelah diproduksi. Sedangkan, *soft microprocessor* ialah suatu *microprocessor core* yang dapat diimplementasikan hanya dengan menggunakan *logic synthesis*. *Logic synthesis* adalah proses yang mengubah spesifikasi abstrak dari perilaku rangkaian yang diinginkan menjadi implementasi desain gerbang logika. Untuk melakukan ini, pengguna menentukan perilaku yang diinginkan dalam *Hardware Description Language*, seperti VHDL.

Terdapat sejumlah *tradeoff* yang perlu dipertimbangkan sebelum menggunakan suatu *soft microprocessor core*(Tong et al., 2006). Dari sisi pro, ada beberapa aspek. Pertama-tama, pembagian kerja antara *CPU* dan *FPGA* lebih mudah dicapai jika pengguna membandingkannya dengan *hard microprocessor core*. Kedua, *soft microprocessor* bersifat portabel kecuali jika menggunakan *vendor-specific microprocessor core*. Ketiga, pengguna dapat dengan bebas mengkonfigurasi fitur-fitur *microprocessor core* sesuai kebutuhan. Namun, sebagai kekurangannya, kinerja *soft microprocessor* lebih rendah daripada *hard microprocessor core*. Meskipun ada keterbatasan kinerja, *soft microprocessor* RISC-V masih mampu menyelesaikan komputasi dengan andal. Selain itu, *soft microprocessor* RISC-V juga sangat fleksibel karena pengguna dapat dengan bebas memodifikasinya sebab sifatnya yang *open source*.

Pengguna perlu melakukan sejumlah tahap untuk menggunakan *soft microprocessor core* dalam suatu *FPGA*. Pertama-tama, pengguna mendesain arsitektur RISC-V *core* dengan *HDL* atau *schematic*. Setelah itu, dilanjutkan dengan *logic synthesis*, yaitu proses translasi *HDL* ke dalam bentuk gerbang logika. Pengguna kemudian menggunakan Hasil sintesis tersebut untuk *functional simulation*. Dalam simulasi ini, pengguna dapat menguji apakah rangkaian dapat mengeksekusi logika seperti yang diinginkan atau tidak. Jika arsitektur berperilaku seperti yang diharapkan, pengguna mengunggah RISC-V core ke FPGA. Ada beberapa langkah dalam proses pengunggahan *FPGA*: *Technology Mapping*,

routing & placement, dan *bitstream* generation. Setelah *compiler* menghasilkan *bitstream*, *USB Blaster* mengunggah *bitstream* ke *FPGA*. Sebagai tambahan, pengguna dapat melakukan *timing analysis* setelah proses *routing & placement*.

Kemudian, untuk menjalankan program high-level pada *soft microprocessor core* RISC-V, pemrogram perlu membuat kode bahasa mesin yang sesuai dengan RISC-V ISA(Poorhosseini et al., 2020). Pemrogram dapat menggunakan *toolchain* yang menghasilkan program Assembly. Setelah pemrogram mengkodekan instruksi *high-level* dalam bahasa C atau bahasa lain, *toolchain* GCC dapat menghasilkan keluaran baik dalam bahasa Assembly atau bentuk biner. Pengguna kemudian dapat menggunakan program biner ini untuk mikroprosesor RISC-V, baik dalam simulasi maupun pada target *FPGA*.

1.2 Permasalahan

Dalam pengembangan prosesor, banyak *Instruction Set Architecture* (*ISA*) dan *microarchitecture* yang sudah dipantenkan. Pengguna membutuhkan suatu lisensi komersial untuk menggunakan *IP* prosesor tradisional, misalnya yang dikembangkan oleh *vendor* Arm, Synopsis, dan Candence. Lisensi *IP* biasanya mencakup *ISA*, *microarchitecture*, dan garansi. Biaya yang dibutuhkan tidak kecil, dimulai dari $75.000 untuk *IP* yang dikembangkan ArmDahad (2018). Akibatnya banyak pihak yang tidak dapat mengakses arsitektur komputer komersial, khususnya di bidang Pendidikan, untuk kepentingan pembelajaran. Fenomena ini menjadi salah satu aspek dibutuhkannya *Instruction Set Architecture open standard* yang dapat digunakan semua pihak dengan bebas(Asanović and Patterson, 2014).

Selain dari faktor lisensi, juga terdapat permasalahan mengenai biaya dan proses produksi(Asicnorth, 2021). Untuk memproduksi suatu *Application Specific IC* (*ASIC*), termasuk *IC* untuk suatu *microprocessor*, dibutuhkan produksi dalam skala massal dengan biaya yang tak kalah besar. Dalam produksi *ASIC* dibutuhkan perencanaan yang matang karena sirkuit tidak bisa diubah jika terdapat kesalahan. Produsen biasanya juga hanya menerima pesanan dalam kuantitas besar, sehingga tidak cocok untuk pembelian dalam skala kecil. Alhasil, *design flow ASIC* cenderung rumit, mulai dari desain logika hingga fabrikasi *silicon*. Oleh karena itu, pada kasus umumnya, tidak digunakan *IC* spesifik, terutama untuk kepentingan edukasi dan *prototyping*. Untuk penggantinya, digunakan suatu *IC reprogrammable*.

1.3 Batasan Masalah

Untuk memfokuskan topik yang diangkat, penulis melakukan pembatasan masalah. Batasan-batasan tersebut, diantaranya:

1. *Instruction Set Architecture* yang digunakan yakni *RV32I*.
2. *Soft processor core* yang dirancang dapat menjalankan semua instruksi kecuali instruksi yang berkaitan dengan *Environment Call*, *CSSR*, dan *FENCE*.
3. *Resource Usage* yang menjadi acuan yaitu jumlah *LUT, Register*, dan *Memory Bits* hasil sintesis.
4. Logika prosesor diverifikasi dengan melakukan simulasi *ModelSim* dan pengujian menggunakan *FPGA board*.
5. Program yang digunakan untuk pengujian merupakan program *low-level RISC-V Assembly* yang diubah ke dalam bentuk heksadesimal dan ditulis secara *hardcode*

ke dalam *program memory*.

1.4 Tujuan

Adapun tujuan dari penilitian ini, diantaranya:

1. Merancang arsitektur prosesor yang dapat menjalankan instruksi dalam *Instruction Set Architecture* RV32I, kecuali `CSSR`, `FENCE`, dan `ECALL`.
2. Memodelkan arsitektur prosesor tersebut ke dalam bentuk *soft processor core* menggunakan bahasa pemodelan *VHDL*.
3. Mensintesis *processor core* yang sudah terverifikasi ke dalam sebuah *Field Programmable Gate Array*.

1.5 Manfaat

Sedangkan, manfaat dari penelitian ini, diantaranya:

1. Menjadi sumber pembelajaran mengenai arsitektur dan organisasi komputer.
2. Menambah wawasan mengenai *Instruction Set Architecture open-standard*, terutama RISC-V.
3. Menurunkan biaya yang dibutuhkan dalam penggunaan *co-processor* untuk desain *FPGA*.

[Halaman ini sengaja dikosongkan]

BAB II
TINJAUAN PUSTAKA

2.1 Penelitian Terdahulu

Penelitian ini mengembangkan hasil yang ditemukan pada sejumlah penelitian terdahulu. Penulis memperbaiki dan mengembangkan *soft processor core* dalam penelitian berjudul *Understanding the Need For An Open ISA and Implementation of A RISC-V Core*. Penulis juga mengembangkan *processor core single-cycle* pada penelitian Kerja Praktik sebelumnya yang berjudul *Synthesizable Single-Cycle FPGA Soft Processor Core*. Selain itu, *Instruction Set Architecture* RISC-V yang digunakan merupakan sebuah *ISA open standard* yang deskripsinya terdapat dalam RISC-V *ISA Manual*.

2.1.1 *Understanding the Need For An Open ISA and Implementation of A RISC-V Core*

Maestro merupakan sebuah *core* prosesor dengan *pipeline* lima tahap berbasiskan RV32I *Instruction Set Architecture* yang terinspirasi oleh buku *Computer Organization and Design* RISC-V *Edition* karangan John Hennesy dan David Patterson (Vitor Rafael Chrisóstomo, 2018). Proyek ini ditujukan untuk keperluan akademis dan dirancang di Federal University of Rio Grande do Norte di Brazil. Perancang dari *core* prosesor ini tidak bertujuan untuk bekompetisi dengan *core* lainnya yang memiliki implementasi kompleks. *core* prosesor ini dibuat untuk keperluan edukasi tentang RISC-V, *ISA*, dan desain prosesor. Proyek ini menggunakan VHDL sebagai *Hardware Description Language*.

Dalam penelitian ini, penulis telah menemukan sejumlah kekurangan dari *soft processor* Maestro. Pertama, penulis menemukan sebuah masalah dengan *flushing unit* ketika penulis mensimulasikan sejumlah instruksi menggunakan simulator ModelSim. Suatu kesalahan terjadi karena terdapat *timing error* ketika *flushing unit* mendeteksi sebuah *jump* atau *branch*. Selebihnya, pada waktu pengembangan, terdapat masalah *timing* pada *processor core* tersebut, sehingga tidak dapat dijalankan dalam *FPGA* yang aktual. Terakhir, terdapat sejumlah instruksi yang tidak diimplementasikan, contohnya *AUIPC*.

2.1.2 *Synthesizable Single-Cycle FPGA Soft Processor Core*

Penelitian ini merupakan proyek sebelumnya yang dilakukan penulis dalam kegiatan Kerja Praktik di Laboratorium Mikroelektronika dan Sistem Tertanam ITS (Aaron Elson, 2022). Proyek ini mengimplementasikan sebuah *soft processor single-cycle* berbasis RISC-V *Instruction Architecture*. Prosesor *single-cycle* hasil rancangan mampu menjalankan semua instruksi dalam RV32I *ISA* kecuali instruksi-instruksi yang berkaitan dengan kapabilitas *multithreading*. *Soft processor single-cycle* ini memerlukan 3.712 *logic elements*, 1.473 *registers*, dan 16.384 *memory bits*. Frekuensi maksimal dari *soft processor* ini yakni 46.6 MHz. Dalam tahap pengujian, digunakan sebuah program deret Fibonacci yang ditampilkan melalui *seven segments display* dan melalui komunikasi *UART*.

2.2 RISC-V *Instruction Set Architecture*

RISC-V adalah *Instruction Set Architecture* (*ISA*) berdasarkan prinsip-prinsip *Reduced Instruction Set Computer* (*RISC*). Tidak seperti kebanyakan desain ISA lainnya, ISA RISC-V disediakan di bawah lisensi *open source* yang tidak memerlukan biaya untuk digunakan(Waterman et al., 2014). Sejumlah perusahaan menawarkan atau telah mengumumkan perangkat keras RISC-V, tersedia sistem operasi open source dengan dukungan RISC-V dan set instruksi didukung di beberapa rangkaian perangkat lunak populer.

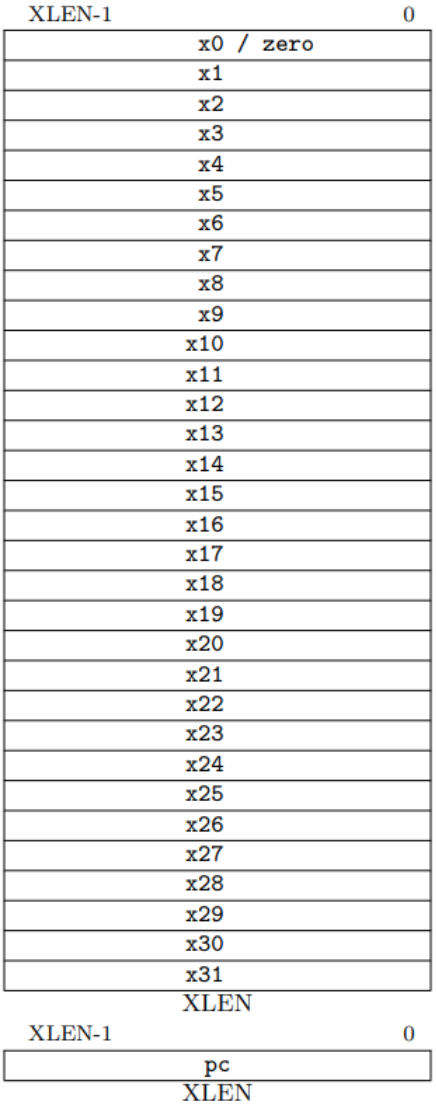
2.2.1 RISC-V *Length Encoding*

Gambar 2.1: *Length encoding dalam RISC-V*(Waterman et al., 2014)

Gambar 2.1 menggambarkan pengkodean panjang pada RISC-V *ISA*. *RISC-V Base ISA* memiliki instruksi 32-bit dengan panjang tetap yang harus di-*align* dengan *boundary* 32-bit. Namun, skema pengkodean RISC-V standar juga dirancang untuk mendukung ekstensi *ISA* dengan instruksi panjang yang variabel, di mana setiap instruksi dapat berupa sejumlah paket instruksi 16-bit dan paket secara yang di-*align* pada *boundary* 16-bit.

2.2.2 *Programmer's Model Pada* RISC-V *ISA*

Gambar 2.2 menggambarkan variabel-variabel yang dapat diakses *user* atau pemrogram dalam RISC-V *ISA*. Variabel yang disimpan dalam *register files* ini dapat diakses oleh pemrogram tanpa harus mengetaui detail dari arsitektur atau bagaimana *hardware* prosesor bekerja. Panjang `XLEN` merupakan panjang bit dari register-register tersebut. Terdapat 31 *general-purpose register* dimulai dari `x1` hingga `x32`, yang dapat menyimpan suatu nilai *integer*. Sedangkan, register `x0` selalu terhubung dengan nilai konstan 0. Tidak ada fungsi khusus atau perilaku yang berbeda antara register selain `x0`. Tetapi, terdapat konvensi tertentu dimana `x1` biasanya digunakan sebagai penyimpanan *return address*. Pada varian RISC-V RV32I, panjang setiap register yaitu 32 *bits* sedangkan pada varian RV64I panjangnya 64 *bits*. Selain 32 *register* tersebut, terdapat register *Program Counter* yang *user-visible*. Pada varian RV32E, yang merupakan varian RISC-V *embedded*, hanya terdapat 16 *registers*.



Gambar 2.2: *User-level base integer register state* pada RISC-V *ISA*(Waterman et al., 2014)

2.2.3 RV32I *ISA*

RV32I Base Instruction Set

imm[31:12]					rd	0110111	LUI
imm[31:12]					rd	0010111	AUIPC
imm[20|10:1|11|19:12]					rd	1101111	JAL
imm[11:0]			rs1	000	rd	1100111	JALR
imm[12|10:5]		rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]		rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]		rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]		rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]		rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]		rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]			rs1	000	rd	0000011	LB
imm[11:0]			rs1	001	rd	0000011	LH
imm[11:0]			rs1	010	rd	0000011	LW
imm[11:0]			rs1	100	rd	0000011	LBU
imm[11:0]			rs1	101	rd	0000011	LHU
imm[11:5]		rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]		rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]		rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]			rs1	000	rd	0010011	ADDI
imm[11:0]			rs1	010	rd	0010011	SLTI
imm[11:0]			rs1	011	rd	0010011	SLTIU
imm[11:0]			rs1	100	rd	0010011	XORI
imm[11:0]			rs1	110	rd	0010011	ORI
imm[11:0]			rs1	111	rd	0010011	ANDI
0000000		shamt	rs1	001	rd	0010011	SLLI
0000000		shamt	rs1	101	rd	0010011	SRLI
0100000		shamt	rs1	101	rd	0010011	SRAI
0000000		rs2	rs1	000	rd	0110011	ADD
0100000		rs2	rs1	000	rd	0110011	SUB
0000000		rs2	rs1	001	rd	0110011	SLL
0000000		rs2	rs1	010	rd	0110011	SLT
0000000		rs2	rs1	011	rd	0110011	SLTU
0000000		rs2	rs1	100	rd	0110011	XOR
0000000		rs2	rs1	101	rd	0110011	SRL
0100000		rs2	rs1	101	rd	0110011	SRA
0000000		rs2	rs1	110	rd	0110011	OR
0000000		rs2	rs1	111	rd	0110011	AND
0000	pred	succ	00000	000	00000	0001111	FENCE
0000	0000	0000	00000	001	00000	0001111	FENCE.I
000000000000			00000	000	00000	1110011	ECALL
000000000001			00000	000	00000	1110011	EBREAK
csr			rs1	001	rd	1110011	CSRRW
csr			rs1	010	rd	1110011	CSRRS
csr			rs1	011	rd	1110011	CSRRC
csr			zimm	101	rd	1110011	CSRRWI
csr			zimm	110	rd	1110011	CSRRSI
csr			zimm	111	rd	1110011	CSRRCI

Gambar 2.3: *32-bit* RISC-V *Base Integer Instruction Set Architecture*(Waterman et al., 2014)

2.2.4 Tipe-Tipe Instruksi dalam RV32I *ISA*

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

Gambar 2.4: Perbedaan antara tipe-tipe instruksi dalam *RV32I ISA*(Waterman et al., 2014)

Dalam RISC-V *Base Integer Instruction Set* terdapat sejumlah tipe inti dari instruksi, yakni tipe R, I, S, dan U. Panjang dari setiap instruksi dalam RV32I *ISA* yakni 32-bit yang di-*align* ke *byte* memori dengan panjang 4-*byte*. Pada semua tipe instruksi, terdapat regularitas yang diterapkan. Contohnya, *register source* `rs1` dan `rs2` serta *register destination* `rd` memiliki posisi yang sama dalam instruksi. *Immidiate* yang disimpan dalam instruksi selalu diproses sebagai nilai *sign-extended*. Nilai *sign-bit* dari *immediate* selalu disimpan pada bit ke-31.

Integer Computational Instructions

Sebagian besar instruksi komputasional integer dalam RV32 memiliki panjang bit `XLEN` yakni 32-bit. Instruksi komputasional integer dikodekan sebagai tipe *register-immediate* dengan tipe I atau *register-register* dengan tipe R. Destinasi dari instruksi yakni `rd` untuk kedua tipe instruksi. Instruksi komputasional integer tidak menghasilkan *exception* jika terjadi *overflow* atau sejenisnya. Pengecekan *overflow* dapat dilakukan dengan serangkaian instruksi yang menghasilkan *pseudofunction*.

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
I-immediate[11:0]	src	ADDI/SLTI[U]	dest	OP-IMM
I-immediate[11:0]	src	ANDI/ORI/XORI	dest	OP-IMM

Gambar 2.5: Pengkodean tipe instruksi komputasional integer *register-immediate* tipe ADDI, SLTI (*unsigned*), ANDI, ORI, dan XORI (Waterman et al., 2014).

Integer Register-Immediate Instructions

1. ADDI merupakan instruksi yang menambahkan sign-extended 12-bit langsung ke register rs1. *Overflow* aritmatika diabaikan dan hasilnya adalah 32-bit terandah yang rendah dari hasilnya. `ADDI rd, rs1, 0` digunakan untuk mengimplementasikan *pseudoinstruction* assembler `MV rd, rs1`.

2. ANDI, ORI, XORI memiliki pengkodean yang mirip dengan ADDI seperti pada Gambar 2.5. Ketiga instruksi tersebut merupakan operasi logika yang melakukan

bitwise AND, OR, dan XOR pada register `rs1` dan *sign-extended* 12-bit *immediate* dan menempatkan hasilnya di destinasi `rd`. Instruksi NOT dapat diterapkan dengan *pseudoinstruction* `XORI rd, rs1, -1`, yang menghasilkan nilai yang sama seperti instruksi `NOT rd, rs`.

3. SLTI (*set less than immediate*) meletakkan nilai 1 pada *register destination* `rd` pada saat `rs1` kurang dari nilai *sign-extended immediate* ketika kedua bilangan tersebut diperlakukan sebagai bilangan *signed*. Jika tidak, dituliskan nilai 0 pada `rd`. Terdapat operasi serupa SLTIU yang dapat membandingkan nilai *unsigned*. Instruksi `SLTIU rd,rs1, 1` dapat digunakan untuk menerapkan *pseudoinstruction* `SEQZ rd,rs`.

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	imm[4:0]	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	shamt[4:0]	src	SLLI	dest	OP-IMM
0000000	shamt[4:0]	src	SRLI	dest	OP-IMM
0100000	shamt[4:0]	src	SRAI	dest	OP-IMM

Gambar 2.6: Pengkodean tipe instruksi komputasional integer *register-immediate* tipe SLLI, SRLI, dan SRAI (Waterman et al., 2014).

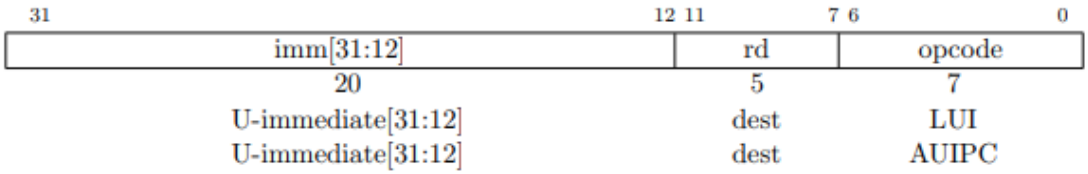
4. *Shifts* (SLLI, SRLI, SRAI) merupakan operasi *shifting* sejumlah nilai *immediate* dan memiliki tipe instruksi I. *Operand* di-*shift* pada `rs1` dan jumlah *shift*-nya yakni 5 bit terendah dari *Immediate field*. SLLI merupakan *logical left shift*. SRLI merupakan *logical right shift*. Dan SRAI merupakan *arithmetic right shift*.

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
I-immediate[11:0]	src	ADDI/SLTI[U]	dest	OP-IMM
I-immediate[11:0]	src	ANDI/ORI/XORI	dest	OP-IMM

Gambar 2.7: Pengkodean tipe instruksi komputasional integer *register-immediate* tipe ADDI, SLTI (*unsigned*), ANDI, ORI, dan XORI (Waterman et al., 2014).

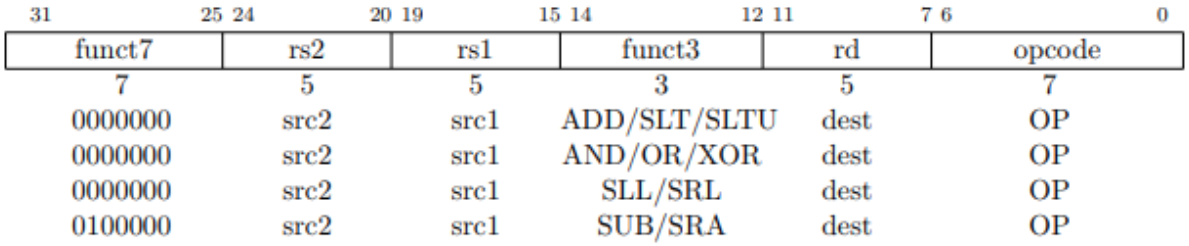
5. LUI (*load upper immediate*) digunakan untuk menyimpan suatu konstanta 32-bit dan menggunakan format tipe U. LUI menempatkan nilai U-*immediate* di 20 bit teratas dari *destination register* `rd`, mengisi nilai 12 bit terendah sebagai 0.

6. AUIPC (*add upper immediate to PC*) digunakan untuk membangun suatu alamat PC-*relative* dan menggunakan tipe pengkodean U. AUIPC membentuk *offset* 32-bit dari *immediate* 20-bit, mengisi 12 bit terendah sebagai 0. *Offset* ini kemudian dijumlahkan ke PC dan hasilnya disimpan ke dalam `rd`.



31 — 12	11 — 7	6 — 0
imm[31:12]	rd	opcode
20	5	7
U-immediate[31:12]	dest	LUI
U-immediate[31:12]	dest	AUIPC

Gambar 2.8: Pengkodean tipe instruksi komputasional integer *register-immediate* tipe LUI dan AUIPC. (Waterman et al., 2014).

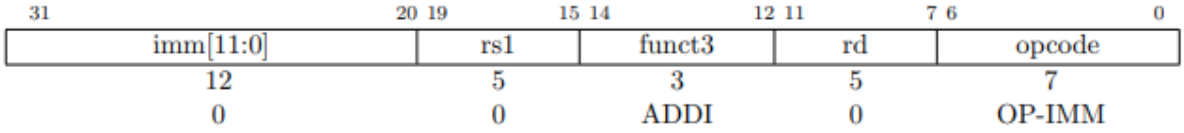


31 — 25	24 — 20	19 — 15	14 — 12	11 — 7	6 — 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	src2	src1	ADD/SLT/SLTU	dest	OP
0000000	src2	src1	AND/OR/XOR	dest	OP
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SUB/SRA	dest	OP

Gambar 2.9: Pengkodean tipe instruksi komputasional integer *register-register* (Waterman et al., 2014).

Integer Register-Register Operations Instruksi tipe R merupakan operasi antara register-register. Operand `rs1` dan `rs2` dari operasi tipe ini yakni berasal dari register, bukan *immediate*. `Funct7` dan `funct3` digunakan untuk menentukan tipe operasi.

ADD dan SUB melakukan operasi adisi dan substraksi. *Overflow* diabaikan dan yang disimpan yakni 32-bit terendah ke dalam *register destination* `rd`. SLT dan SLTU melakukan operasi yang *signed* dan *unsigned*. SLL, SRL, dan SRA melakukan operasi *logical left*, *logical right*, dan *arithmetic right shifting*. AND, OR, dan XOR melalukan operasi logika *bitwise*.



31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
0	0	ADDI	0	OP-IMM

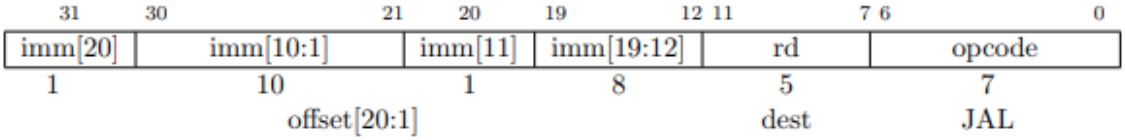
Gambar 2.10: pengkodean operasi *pseudoinstruction* NOP menggunakan instruksi ADDI (Waterman et al., 2014).

NOP Instruction Instruksi NOP tidak mengubah status apapun, kecuali menjumlahkan nilai PC ke alamat berikutnya. NOP dapat diterapkan menggunakan *pseudoinstruction* yang dibangun menggunakan instruksi `ADDI x0,x0,0`.

Control Transfer Instructions

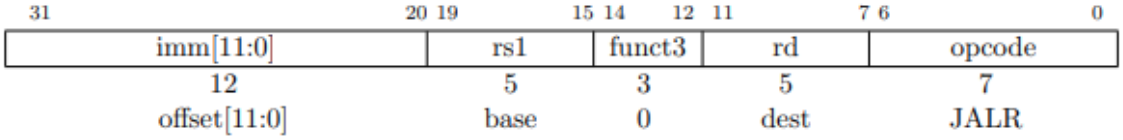
Dalam RISC-V, terdapat dua jenis instruksi *control transfer*: *unconditional jump* dan *conditional jump*. Tidak terdapat *visible delay slots* yang terlihat secara arsitektural

dalam RV32I.



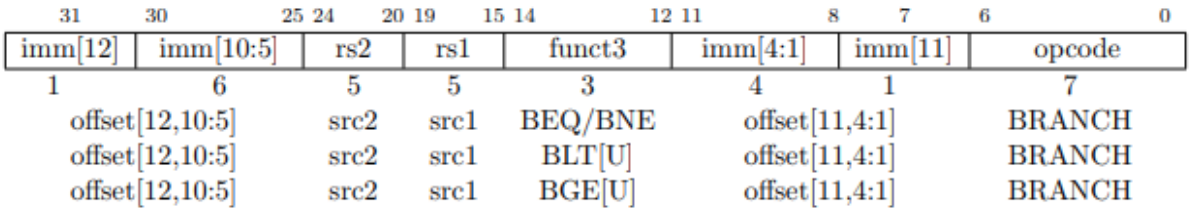
Gambar 2.11: pengkodean operasi JAL (Waterman et al., 2014).

Unconditional Jump Instruksi *Conditional Jump* pada RV32I yakni Jump and Link (JAL) yang memiliki tipe J. J *immediate* mengkodekan *offset* dalam kelipatan 2-*byte*. *Offset* ini di-*sign-extend* dan dijumlahkan ke PC untuk membentuk *jump target address*. JAL dapat melakukan lompatan dengan *range* sekitar 1 MiB. JAL menyimpan address dari instruksi PC+4 ke register `rd`. Konvensi *software* menggunakan `x1` sebagai *return address register* dan `x5` sebagai *alternate link register*. Juga terdapat *Plain unconditional jump* yang dapat diterapkan sebagai *pseudoinstruction* JAL dengan `rd=0`. *Pseudoinstruction* J ini tidak menyimpan *return address*.



Gambar 2.12: pengkodean operasi JALR (Waterman et al., 2014).

Selain JAL, terdapat operasi *jump* yang *indirect* yakni JALR. Operasi bertipe I ini menggunakan *target address* yang merupakan penjumlahan dari 12-bit *immedeate* dengan *operand* register `rs1` yang *signed*. *LSB* dari hasil diset ke 0 Alamat selanjutnya dari instruksi atau PC+4 dituliskan ke register `rd`. Register `x0` dapat digunakan sebagai destinaasi jika hasilnya tidak diperlukan.

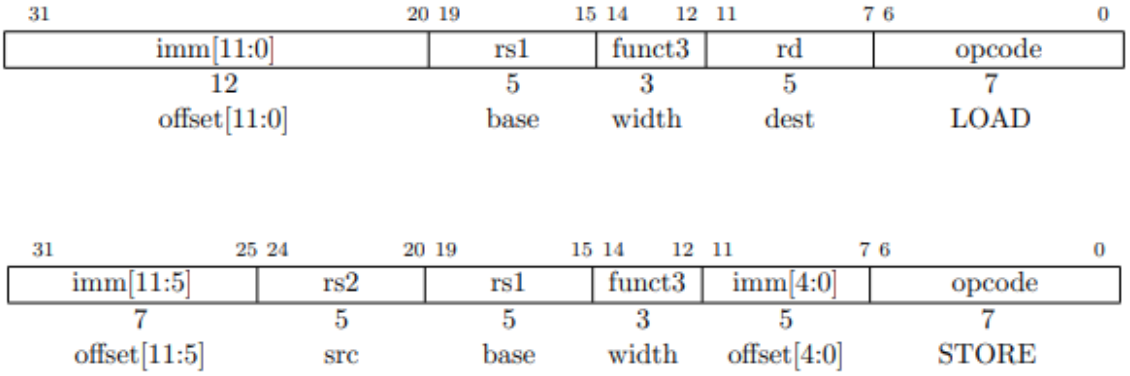


Gambar 2.13: pengkodean operasi-operasi *branching* kondisional(Waterman et al., 2014).

Conditional Branches Semua instruksi *branch* memiliki format instruksi tipe B. Nilai *immediate* 12-bit dalam instruksi merupakan *offset* kelipatan 2 yang *signed*. Untuk mendapatkan *target address*, nilai tersebut ditambahkan ke PC. *Range* dari instruksi *branch*

yakni sekitar 4 KiB. Semua instruksi *branch* membandingkan dua register sebelum melakukan *branching*. BEQ dan BNE melakukan *branching* jika `rs1` dan `rs2` nilainya *equal* dan *not equal*.

Load and Store Instructions

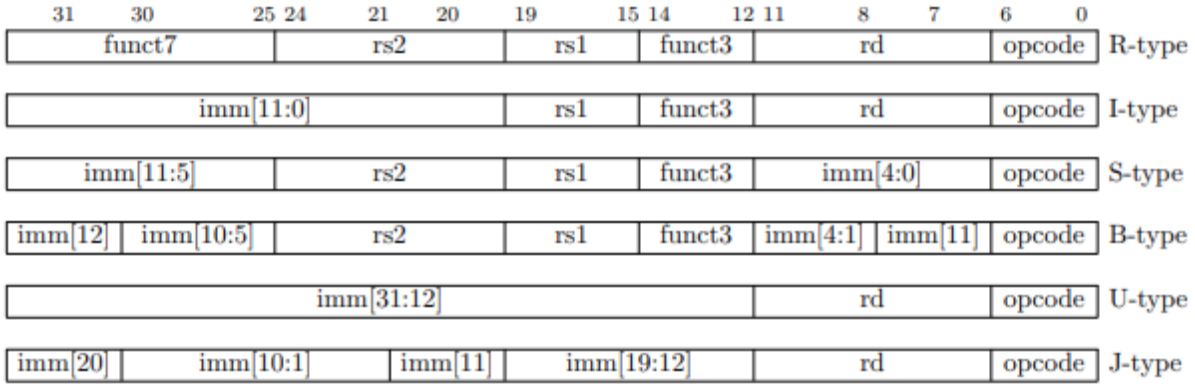


Gambar 2.14: pengkodean operasi *Load* dan *Store* (Waterman et al., 2014).

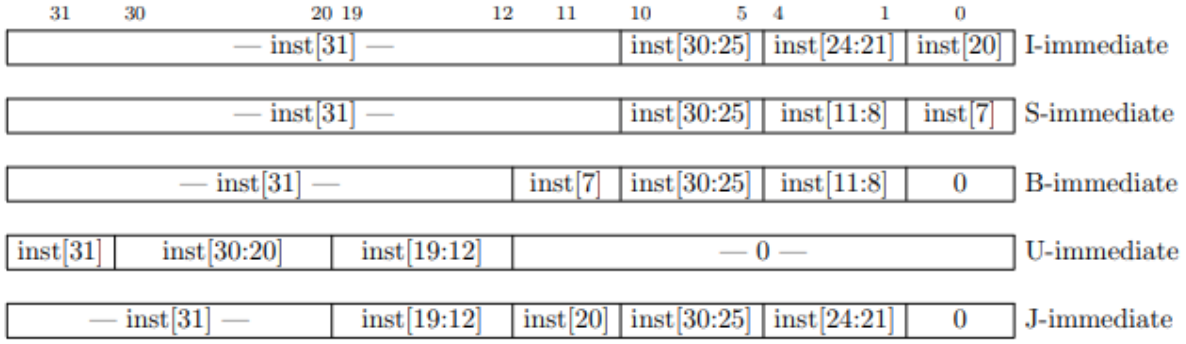
Conditional Branches Dalam *ISA* RV32I, hanya instruksi *Load* dan *Store*-lah yang menggunakan *data memory* dan instruksi *arithmetic* hanya menggunakan register *CPU*. RV32I memberikan *user address space* 32-bit yang *little-endian*. *Load* dan *store* melakukan transfer antara register dan memori. *Load* dikodekan sebagai tipe I dan tipe S. Alamat efektif dari instruksi didapatkan dari penjumlahan `rs1` ke *offset* 12-bit yang *sign-extended*. *Load* menyalin nilai dari memori ke register `rd`. *Store* menyalin nilai dalam register `rs2` ke memori.

1. LW melakukan *load* nilai 32-bit dari memori ke `rd`.
2. LH melakukan *load* nilai 16-bit dari memori dan kemudian melakukan *sign extension* menjadi 32-bit ke `rd`.
3. LHU melakukan *load* nilai 16-bit dan melakukan *zero extgend* menjadi nilai 32-bit ke `rd`.
4. LB dan LBU sama seperti sebelumnya namun untuk nilai 8-bit.
5. SW, SH, dan SB menyimpan nilai 32-bit, 16-bit, dan 8-bit dari memori ke *low bits* dari register `rs2` ke memori.

2.2.5 Varian *Immediate Encoding* RISC-V



Gambar 2.15: pengkodean *immediate* pada *base instruction* RISC-V (Waterman et al., 2014).



Gambar 2.16: Tipe *immediate* yang dihasilkan oleh instruksi-instruksi RISC-V.(Waterman et al., 2014).

Terdapat dua tipe tambahan dari format B dan J tergantung dari bagaimana *immediate* diperlakukan. Ini dapat dilihat dalam Gambar 2.15. Perbedaan dari format S dan B yakni *immediate* 12-bit digunakan untuk *offset branch* dengan kelipatan 2 dalam format B. Juga terdapat perbedaan yang mirip antara format U dan J dimana perbedaannya yakni pada *shifting* 1 bit.

2.2.6 Instruksi-Instruksi dalam RV32I *Instruction Set Architecture*

1. LUI

Tabel 2.1: Format instruksi LUI.

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
imm[31:12]					rd	01101	11

Format: `lui rd,imm`

Deskripsi: Membangun konstanta 32-bit dan menggunakan format tipe U. LUI meletakkan nilai *immediate* U pada 20 bit teratas *register destination* `rd`, mengisi 12 bit terendahnya dengan nol.

Implementasi: `x[rd] = sign.extend(immediate[31:12] << 12)`

2. AUIPC

Tabel 2.2: Format instruksi AUIPC.

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
imm[31:12]					rd	00101	11

Format: `auipc rd,imm`

Deskripsi: Membangun alamat *PC-relative* dan menggunakan format tipe U. AUIPC membentuk *offset* 32-bit dari 20-bit *immediate* tipe U, menjumlahkan *offset* ini ke *PC*, kemudian menyimpan hasilnya ke register `rd`.

Implementasi: `x[rd] = pc + sign.extend(immediate[31:12] << 12)`

3. JAL

Tabel 2.3: Format instruksi JAL.

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
offset[20—10:1—11			19:12]		rd	11011	11

Format: `jal rd,offset`

Deskripsi:*Jump* ke alamat dan meletakkan *return address* di `rd`.

Implementasi: `x[rd] =pc+4;pc += sign.extend(offset)`

4. JALR

Tabel 2.4: Format instruksi JALR.

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
offset[11:0]			rs1	000	rd	11001	11

Format: `jalr rd,rs1,offset`

Deskripsi: *Jump* ke alamat dan meletakkan *return address* di `rd`.

Implementasi:
```
t = pc+4
pc = (x[rs1]+sign.extend(offset)& ~ 1
x[rd]=t
```

5. BEQ

Tabel 2.5: Format instruksi BEQ.

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
offset[12—10:5]		rs2	rs1	000	offset[4:1—11]	11000	11

Format: `beq rs1,rs2,offset`

Deskripsi: Melakukan *branch* jika `rs1` dan `rs2` bernilai sama.

Implementasi: `if (rs1 == rs2) pc += sign.extend(offset)`

6. BNE

Tabel 2.6: Format instruksi BNE.

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
offset[12—10:5]		rs2	rs1	001	offset[4:1—11]	11000	11

Format: `bne rs1,rs2,offset`

Deskripsi: Melakukan *branch* jika `rs1` dan `rs2` bernilai tidak sama.

Implementasi: `if (rs1 != rs2) pc += sign.extend(offset)`

7. BLT

Tabel 2.7: Format instruksi BLT.

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
offset[12—10:5]		rs2	rs1	100	offset[4:1—11]	11000	11

Format: `blt rs1,rs2,offset`

Deskripsi: Melakukan *branch* jika `rs1` kurang dari `rs2`, menggunakan komparasi *signed*.

Implementasi: `if (rs1 < signed rs2) pc += sign.extend(offset))`

8. BGE

Tabel 2.8: Format instruksi BGE.

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
offset[12—10:5]		rs2	rs1	101	offset[4:1—11]	11000	11

Format: `bge rs1,rs2,offset`

Deskripsi: Melakukan *branch* jika `rs1` lebih dari `rs2`, menggunakan komparasi *signed*.

Implementasi: `if (rs1 > signed rs2) pc += sign.extend(offset))`

9. BLTU

Tabel 2.9: Format instruksi BLTU.

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
offset[12—10:5]		rs2	rs1	110	offset[4:1—11]	11000	11

Format: `bltu rs1,rs2,offset`

Deskripsi: Melakukan *branch* jika `rs1` kurang dari `rs2`, menggunakan komparasi *unsigned*.

Implementasi: `if (rs1 < unsigned rs2) pc += sign.extend(offset)))`

10. BGEU

Tabel 2.10: Format instruksi BGEU.

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
offset[12—10:5]		rs2	rs1	111	offset[4:1—11]	11000	11

Format: `lui rd,imm`

Deskripsi: Melakukan *branch* jika `rs1` lebih dari `rs2`, menggunakan komparasi *unsigned*.

Implementasi: `if (rs1 > unsigned rs2) pc += sign.extend(offset)))`

11. LB

Tabel 2.11: Format instruksi LB.

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
offset[11:0]			rs1	000	rd	00000	11

Format: `lb rd,offset(rs1)`

Deskripsi: Memuat nilai 8-bit dari memori dan melakukan *sign-extend* ke 32-bit sebelum menyimpannya dalam register `rd`.

Implementasi: `x[rd]= sign.extend(Memory[x[rs1] + sign.extend(offset)][7:0])`

12. LH

Tabel 2.12: Format instruksi LH.

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
offset[11:0]			rs1	001	rd	00000	11

Format: `lh rd,offset(rs1)`

Deskripsi: Memuat nilai 16-bit dari memori dan melakukan *sign-extend* ke 32-bit sebelum menyimpannya dalam register `rd`.

Implementasi: `x[rd] = sign.extend(Memory[x[rs1] + sign.extend(offset)][15:0])`

13. LW

Tabel 2.13: Format instruksi LW.

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
offset[11:0]			rs1	010	rd	00000	11

Format: `lw rd,offset(rs1)`

Deskripsi: Memuat nilai 32-bit dari memori dan melakukan *sign-extend* ke 32-bit sebelum menyimpannya dalam register `rd`.

Implementasi: `x[rd] = sign.extend(Memory[x[rs1] + sign.extend(offset)][32:0])`

14. LBU

Tabel 2.14: Format instruksi LBU.

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
offset[11:0]			rs1	100	rd	00000	11

Format: `lbu rd,offset(rs1)`

Deskripsi: Memuat nilai 8-bit dari memori dan melakukan *zero-extend* ke 32-bit sebelum menyimpannya dalam register `rd`.

Implementasi: `x[rd] = Memory[x[rs1] + sign.extend(offset)][7:0]`

15. LHU

Tabel 2.15: Format instruksi LHU.

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
offset[11:0]			rs1	101	rd	00000	11

Format: `lhu rd,offset(rs1)`

Deskripsi: Memuat nilai 16-bit dari memori dan melakukan *zero-extend* ke 32-bit sebelum menyimpannya dalam register `rd`.

Implementasi: `x[rd] = Memory[x[rs1] + sign.extend(offset)][15:0]`

16. SB

Tabel 2.16: Format instruksi SB.

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
offset[11:5]		rs2	rs1	000	offset[4:0]	01000	11

Format: `sb rs2,offset(rs1)`

Deskripsi: Menyimpan nilai 8-bit dari bit-bit terendah register `rs2` ke memori.

Implementasi: `Memori[x[rs1] + sign.extend(offset)] = x[rs2][7:0]`

17. SH

Tabel 2.17: Format instruksi SH.

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
offset[11:5]		rs2	rs1	001	offset[4:0]	01000	11

Format: `sh rs2,offset(rs1)`

Deskripsi: Menyimpan nilai 16-bit dari bit-bit terendah register `rs2` ke memori.

Implementasi: `Memori[x[rs1] + sign.extend(offset)] = x[rs2][15:0]`

18. SW

Tabel 2.18: Format instruksi SW.

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
offset[11:5]		rs2	rs1	010	offset[4:0]	01000	11

Format: `sw rs2,offset(rs1)`

Deskripsi: Menyimpan nilai 32-bit dari bit-bit terendah register `rs2` ke memori.

Implementasi: `Memori[x[rs1] + sign.extend(offset)] = x[rs2][31:0]`

19. ADDI

Tabel 2.19: Format instruksi ADDI.

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
imm[11:0]			rs1	000	rd	00100	11

Format: `addi rd,rs1,imm`

Deskripsi: Menjumlahkan 12-bit *immediate* yang sudah di-*sign-extend* ke register `rs1`. *Overflow* dihiraukan dan hasilnya yakni 32-bit terendah.

Implementasi: `x[rd] = x[rs1] + sign.extend(immediate)`

20. SLTI

Tabel 2.20: Format instruksi SLTI.

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
imm[11:0]			rs1	010	rd	00100	11

Format: `slti rd,rs1,imm`

Deskripsi: Meletakkan nilai 1 dalam register `rd` jika `rs1` kurang dari nilai *sign-extend immediate* ketika keduanya diperlakukan sebagai bilangan *signed*. Selain itu, dituliskan 0 ke `rd`.

Implementasi: `x[rd] = x[rs1] <signed sign.extend(immediate)`

21. SLTIU

Tabel 2.21: Format instruksi SLTIU.

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
imm[11:0]			rs1	011	rd	00100	11

Format: `sltiu rd,rs1,imm`

Deskripsi: Meletakkan nilai 1 dalam register `rd` jika `rs1` kurang dari nilai *sign-extend immediate* ketika keduanya diperlakukan sebagai bilangan *unsigned*. Selain itu, dituliskan 0 ke

Implementasi: `x[rd] = x[rs1] <unsigned sign.extend(immediate)`

22. XORI

Tabel 2.22: Format instruksi XORI.

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
imm[11:0]			rs1	100	rd	00100	11

Format: `xori rd,rs1,imm`

Deskripsi: Melakukan *bitwise* XOR pada register `rs1` dan *sign-extend* 12-bit *immediate* dan meletakka hasilnnya ke `rd`.

Implementasi: `x[rd] = x[rs1] ^ sign.extend(immediate)`

23. ORI

Tabel 2.23: Format instruksi ORI.

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
imm[11:0]			rs1	110	rd	00100	11

Format: `ori rd,rs1,imm`

Deskripsi: Melakukan *bitwise* OR pada register `rs1` dan *sign-extend* 12-bit *immediate* dan meletakkan hasilnya ke `rd`.

Implementasi: `x[rd] = x[rs1] | sign.extend(immediate)`

24. ANDI

Tabel 2.24: Format instruksi ANDI.

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
imm[11:0]			rs1	111	rd	00100	11

Format: `andi rd,rs1,imm`

Deskripsi: Melakukan *bitwise* AND pada register `rs1` dan *sign-extend* 12-bit *immediate* dan meletakkan hasilnnya ke `rd`

Implementasi: `x[rd] = x[rs1] & sign.extend(immediate)`

25. SLLI

Tabel 2.25: Format instruksi SLLI.

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
00000	0X	shamt	rs1	001	rd	00100	11

Format: `slli rd,rs1,shamt`

Deskripsi: Melakukan *logical left shift* pada nilai dalam `rs1` dengan nilai *shift* dalam 5 bit-bit terendah dari *immediate*.

Implementasi: `x[rd] = sign.extend(x[rd] = x[rs1] << shamt`

26. SRLI

Tabel 2.26: Format instruksi SRLI.

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
00000	0X	shamt	rs1	101	rd	00100	11

Format: `x[rd] = x[rs1] >>unsigned shamt`

Deskripsi: Melakukan *logical right shift* pada nilai dalam `rs1` dengan nilai *shift* dalam 5 bit-bit terendah dari *immediate*.

Implementasi: `x[rd] = sign.extend(immediate[31:12] << 12)`

27. SRAI

Tabel 2.27: Format instruksi SRAI.

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
01000	0X	shamt	rs1	101	rd	00100	11

Format: `srai rd,rs1,shamt`

Deskripsi: Melakukan *arithmetic right shift* pada nilai dalam `rs1` dengan nilai *shift* dalam 5 bit-bit terendah dari *immediate*.

Implementasi: `x[rd] = x[rs1] >>signed shamt`

28. ADD

Tabel 2.28: Format instruksi ADD.

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
00000	00	rs2	rs1	000	rd	01100	11

Format: `add rd,rs1,rs2`

Deskripsi: Menjumlahkan register `rs1` dan `rs2` dan menyimpan hasilnya dalam `rd`. *Overflow* dihiraukan dan nilai yang disimpan yakni 32-bit terendah.

Implementasi: `x[rd] = x[rs1] + x[rs2]`

29. SUB

Tabel 2.29: Format instruksi SUB.

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
01000	00	rs2	rs1	000	rd	01100	11

Format: `sub rd,rs1,rs2`

Deskripsi: Mengurangi register `rs1` dengan `rs2` dan menyimpan hasilnya dalam `rd`. *Overflow* dihiraukan dan nilai yang disimpan yakni 32-bit terendah.

Implementasi: `x[rd] = x[rs1] - x[rs2]`

30. SLL

Tabel 2.30: Format instruksi SLL.

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
00000	00	rs2	rs1	001	rd	01100	11

Format: `x[rd] = x[rs1] << x[rs2]`

Deskripsi: Melakukan *logical left shift* pada nilai dalam `rs1` dengan nilai *shift* dalam 5 bit-bit terendah dari register `rs2`

Implementasi: `x[rd] = sign.extend(immediate[31:12] << 12)`

31. SLT

Tabel 2.31: Format instruksi SLT.

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
00000	00	rs2	rs1	010	rd	01100	11

Format: `x[rd] = x[rs1] <signed x[rs2]`

Deskripsi: Meletakkan nilai 1 dalam register `rd` jika `rs1` kurang dari nilai `rs2` ketika keduanya diperlakukan sebagai bilangan *signed*. Selain itu, dituliskan 0 ke `rd`.

Implementasi: `x[rd] = x[rs1] <unsigned x[rs2]`

32. SLTU

Tabel 2.32: Format instruksi SLTU.

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
00000	00	rs2	rs1	011	rd	01100	11

Format: `sltu rd,rs1,rs2`

Deskripsi: Meletakkan nilai 1 dalam register `rd` jika `rs1` kurang dari nilai `rs2` ketika keduanya diperlakukan sebagai bilangan *unsigned*. Selain itu, dituliskan 0 ke `rd`.

Implementasi: `x[rd] = x[rs1] <unsigned x[rs2]`

33. XOR

Tabel 2.33: Format instruksi XOR.

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
00000	00	rs2	rs1	100	rd	01100	11

Format: `xor rd,rs1,rs2`

Deskripsi: Melakukan *bitwise* XOR pada register `rs1` dan `rs2` dan meletakkan hasilnya ke `rd`.

Implementasi: `x[rd] = x[rs1] ^ x[rs2]`

34. SRL

Tabel 2.34: Format instruksi SRL.

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
00000	00	rs2	rs1	101	rd	01100	11

Format: `x[rd] = x[rs1] >>u x[rs2]`

Deskripsi: Melakukan *logical right shift* pada nilai dalam `rs1` dengan nilai *shift* dalam 5 bit-bit terendah dari register `rs2`.

Implementasi: `x[rd] = sign.extend(immediate[31:12] << 12)`

35. SRA

Tabel 2.35: Format instruksi SRA.

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
01000	00	rs2	rs1	101	rd	01100	11

Format: `x[rd] = x[rs1] >>s x[rs2]`

Deskripsi: Melakukan *arithmetic right shift* pada nilai dalam `rs1` dengan nilai *shift* dalam 5 bit-bit terendah dari register `rs2`

Implementasi: `x[rd] = sign.extend(immediate[31:12] << 12)`

36. OR

Tabel 2.36: Format instruksi OR.

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
00000	00	rs2	rs1	110	rd	01100	11

Format: `x[rd] = x[rs1] | x[rs2]`

Deskripsi: Melakukan *bitwise* OR pada register `rs1` dan `rs2` dan meletakkan hasilnya ke `rd`.

Implementasi: `x[rd] = sign.extend(immediate[31:12] << 12)`

37. AND

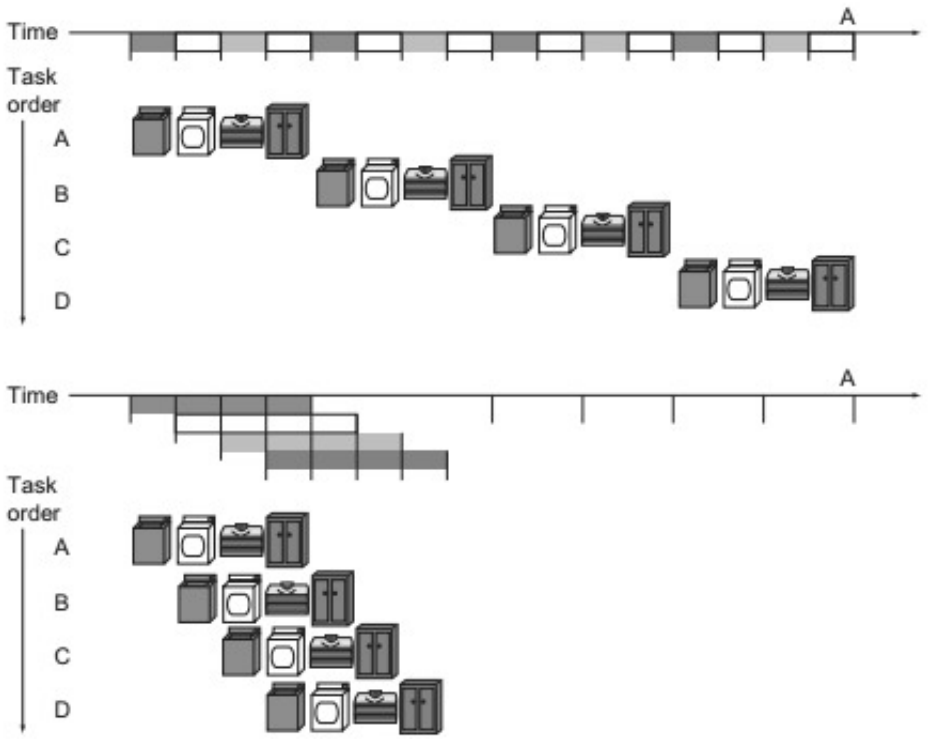
Tabel 2.37: Format instruksi AND.

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
00000	00	rs2	rs1	111	rd	01100	11

Format: `and rd,rs1,rs2`

Deskripsi: Melakukan *bitwise* AND pada register `rs1` dan `rs2` dan meletakkan hasilnya ke `rd`.

Implementasi: `x[rd] = x[rs1] & x[rs2]`



Gambar 2.17: Illustrasi eksekusi sejumlah instruksi dalam sebuah *pipeline*(Patterson and Hennessy, 2017). Diagram pertama merupakan penjalanan instruksi tanpa *pipelining*, dan diagram kedua merupakan penjalan instruksi dengan *pipelining*.

2.2.7 Pipelining

Pipelining merupakan pola *parallelism* yang banyak muncul dalam arsitektur komputer. Dalam suatu *pipeline*, beberapa instruksi dijalankan saling tumpang tindih, menyerupai suatu *assembly line*(Patterson and Hennessy, 2017). Elemen-elemen dari sebuah *pipeline* sering dieksekusi secara paralel atau dalam *mode time-slice*. Beberapa jumlah *buffer storage* sering disisipkan di antara elemen.

Gambar 2.17 menggambarkan cara kerja sebuah *pipeline*. Dalam eksekusi suatu instruksi komputer, dapat dibagi menjadi sejumlah tahap. Misalnya, pada RISC-V terdapat lima tahapan, yakni: *Instruction Fetch*, *Instruction Decode*, *Execution*, *Memory*, dan *Write Back*. Kelima tahap ini memiliki *critical path* masing-masing yang menjadi batasan frekuensi *clock*. Tanpa *pipeline*, semua tahap tersebut harus diselesaikan terlebih dahulu sebelum instruksi kemudian masuk. Sedangkan, jika digunakan suatu *pipeline*, tahap-tahap tersebut dapat dijalankan bersamaan secara paralel. Dengan demikian, *clock* maksimal dari prosesor dapat ditingkatkan, karena yang menjadi *critical path* bukanlah gabungan dari semua tahap lagi, melainkan *datapath* dari tahap yang terpanjang. Metode ini dapat digunakan untuk meningkatkan frekuensi dan *throughput* dari suatu design. Walaupun demikian, dapat terjadi peningkatan *latency* antara desain tanpa *pipeline* dan desain dengan *pipeline*. Hal ini dikarenakan terdapat *overhead delay* yang dibutuhkan untuk menyeimbangkan *delay* antar tahapan *pipeline*.

2.2.8 Soft Microprocessor Core

Soft Microprocessor (juga disebut softcore microprocessor atau soft processor) adalah desain *CPU* yang ditulis dalam bahasa deskripsi logika yang dapat disintesis ke dalam suatu *programmable logic device*(McLoughlin, 2018). Bahasa *HDL* yang biasanya digunakan untuk mendeskripsikan Soft Microprocessor yakni VHDL dan Verilog. Ini dapat diimplementasikan melalui perangkat semikonduktor berbeda yang berisi logika yang dapat diprogram (mis., ASIC, FPGA, CPLD).

2.2.9 *Field-Programmable Gate Array* (*FPGA*)

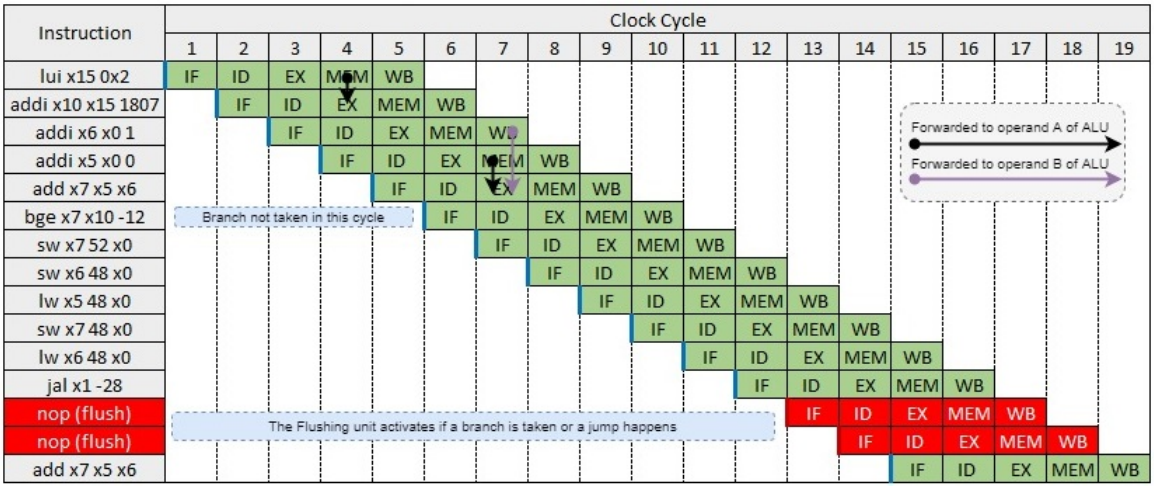
Field-programmable gate array (*FPGA*) adalah *configurable integrated circuit* yang terdiri dari blok logika kombinasional dan flip-flop(Patterson and Hennessy, 2017). Konfigurasi *FPGA* umumnya ditentukan menggunakan *Hardware Description Language* (*HDL*), mirip dengan yang digunakan untuk *Application Specific Integrated Circuit* (*ASIC*). Diagram sirkuit sebelumnya digunakan untuk menentukan konfigurasi, tetapi ini semakin jarang karena munculnya alat otomatisasi desain elektronik.

2.2.10 VHSIC Hardware Description Language (VHDL)

VHSIC Hardware Description Language (VHDL) adalah bahasa deskripsi perangkat keras (*HDL*) yang dapat memodelkan perilaku dan struktur sistem digital pada berbagai tingkat abstraksi, mulai dari tingkat sistem hingga gerbang logika, untuk entri desain, dokumentasi, dan tujuan verifikasi. VHDL merupakan *HDL* pertama yang distandardisasi IEEE. Untuk memodelkan sistem analog dan sinyal campuran, *HDL* standar IEEE berdasarkan VHDL yang disebut VHDL *Analog Mixed-Signal* (secara resmi IEEE 1076.1) telah dikembangkan (Krolikoski, 2011).

2.3 *Hazard* dalam Pipelining

Terdapat situasi dalam *pipelining* dimana instruksi selanjutnya tidak dapat dilakukan pada *clock* berikutnya. Peristiwa ini dinamakan *hazard* dan terdapat tiga tipe dari *hazard*.



Gambar 2.18: Illustrasi *hazard* dan penyelesaiannya pada eksekusi program deret Fibonacci.

2.3.1 Structural Hazard

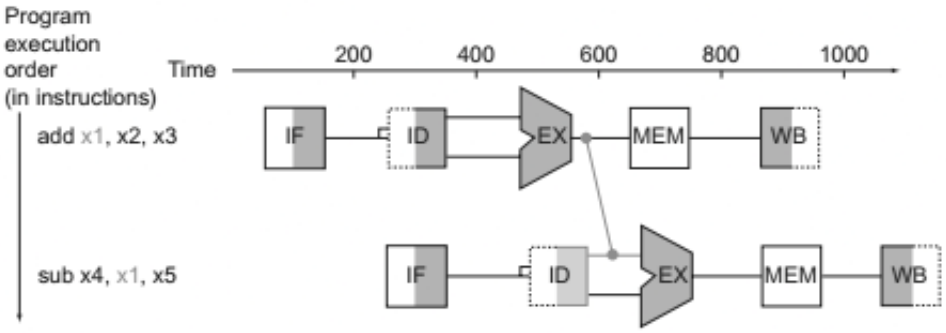
Hazard pertama disebut *structural hazard*. Artinya *hardware* dari *CPU* tidak dapat mendukung kombinasi instruksi yang ingin dijalankan dalam siklus *clock* yang sama. *Structural hazard* dalam *pipeline* terjadi jika penulis menggunakan perangkat *hardware* yang sama oleh dua instruksi pada saat yang bersaman. Untuk mengatasi ini, perlu berhati-hati dalam implementasi *hardware CPU* dengan *pipeline*.

2.3.2 Data Hazard

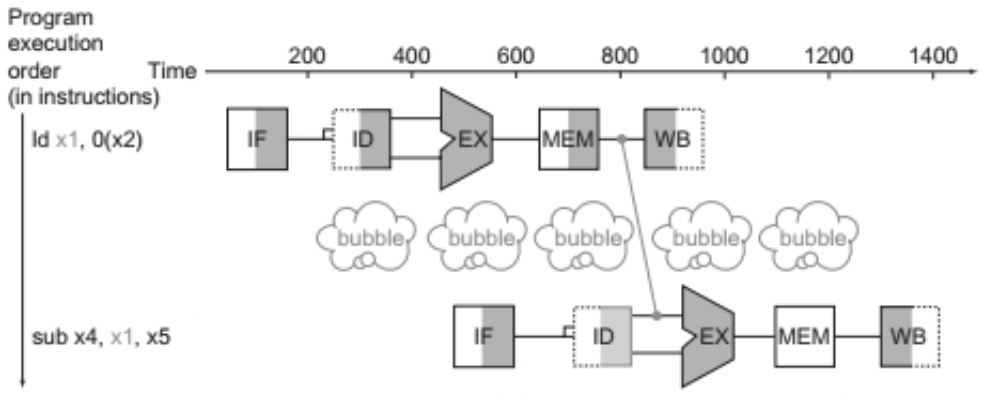
Data Hazard terjadi pada saat *pipeline* harus di-*stall* dan harus menunggu hingga operasi sebelumnya selesai. *Data Hazard* dalam pipeline karena adanya depedensi terhadap suatu instruksi yang muncul lebih awal yang masih dalam proses. Sebagai contoh, ini dapat terjadi dalam instruksi:

```
add x19, x0, x1
sub x2, x19, x3
```

Tanpa suatu metode penanganan, *data hazard* dapat menjadi halangan besar dalam *pipeline*. Ini karena *hazard* mengakibatkan *stall* selama tiga *clock cycle*. Oleh karena itu, perlu dilakukan suatu metode yang dikenal sebagai *forwarding*. Metode ini diterapkan dengan *hardware* tambahan pada *pipeline*.Gambar 2.19 merupakan salah satu metode penanganan *data hazard* dengan menggunakan teknik *forwarding*. *Forwarding unit* meneruskan nilai dari instruksi sebelumnya yang berada pada stage *Memory* atau *Write Back* ke dalam *Arithmetic Logic Unit* pada stage *Execution* saat ini. Selanjutnya, Gambar 2.20 menggambarkan metode penyelesaian *data hazard* dengan teknik *stalling*. Penanganan *hazard* pada instruksi *Load* memerlukan *hazard detection unit* yang melakukan *stall* karena *forwarding unit* tidak bisa meneruskan



Gambar 2.19: Illustrasi *forwarding* dalam *pipeline* (Patterson and Hennessy, 2017).



Gambar 2.20: Pada tipe instruksi tipe R, dibutuhkan *stall* dan *forwarding* (Patterson and Hennessy, 2017)

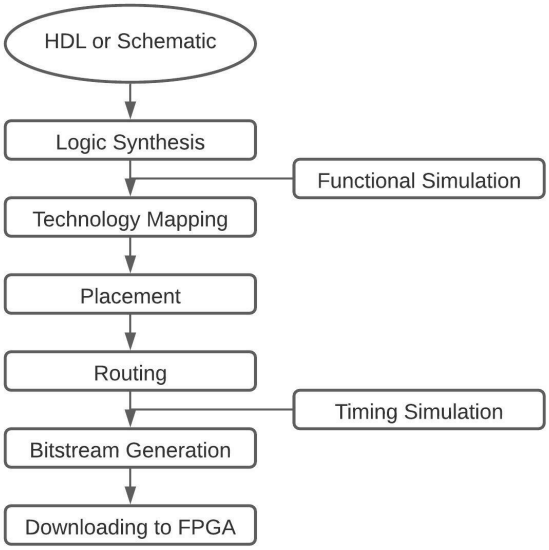
2.3.3 Control Hazard

Tipe terakhir *hazard* dalam RISC-V yakni *Control Hazard*. *Control hazard* juga disebut sebagai *branch hazard*. *Hazard* tipe ini muncul dari keperluan pembuatan keputusan berdasarkan hasil dari instruksi yang sedang diproses. *Hazard* ini terjadi misalnya pada instruksi *Jump* dan *Branch*. Pada saat *Jump* terjadi atau *branch taken*, tahap *pipeline Execution* memberikan mengeluarkan alamat target *program counter*. Masalahnya, instruksi dari *stage* sebelum *Execution*, yakni *Instruction Fetch* dan *Instruction Decode* masih mengandung instruksi dan nilai-nilai sebelum *Jump* dilakukan. Jika *hazard* tidak diperbaiki, maka pada *clock cycle* berikutnya hasil dari tahap *Execution* menjadi keliru. Terdapat cara yang biasanya digunakan untuk mengatasi *Control hazard*, yakni: pengubahan arsitektur prosesor dan implementasi *branch prediction*. Pengubahan arsitektur ditujukan untuk menghapus nilai yang salah pada saat *jump* atau *branch*, yang disebut sebagai *flushing*. Sedangkan, *branch prediction* dilakukan dengan cara memprediksi kapan *branch* atau *jump* terjadi. Selain itu, bisa dilakukan metode *stall* 2 *cycle clock*, dimana prosesor baru melanjutkan instruksi pada saat instruksi *jump* atau *branch* mencapai *stage pipeline* terakhir, yakni *Writeback*.

[Halaman ini sengaja dikosongkan]

BAB III
METODOLOGI

3.1 Metodologi Penelitian

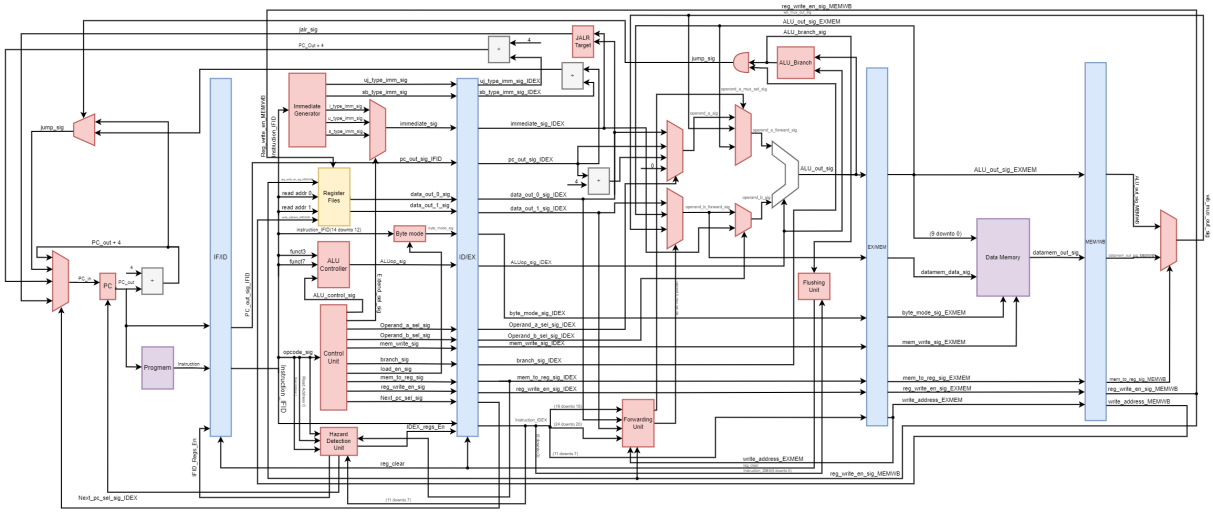


Gambar 3.1: *FPGA Design Methodology* (Singh and Rajawat, 2013).

Dari segi perancangan sistem, dilaksanakan metodologi desain FPGA standar seperti pada Gambar 3.1. Ini meliputi, namun tidak terbatas pada: deskripsi sistem melalui *Hardware Description Language*, *logic synthesis*, *functional simulation*, *technology mapping*, *placement & routing*, *timing simulation*, *bitstream generation*, dan pada akhirnya pengunggahan ke FPGA. Dalam penggunaan HDL, penulis mengabstraksi *behavior* dari sirkuit RISC-V *microprocessor*. Kemudian, penulis melakukan *logic synthesis* terhadap kode HDL tersebut. Sintesis logika mengubah abstraksi yang penulis buat menjadi suatu implementasi dalam bentuk gerbang logika. Hasil dari sintesis logika kemudian penulis gunakan untuk melakukan sebuah *functional simulation*. Simulasi funsional atau simulasi logika adalah penggunaan perangkat lunak simulasi untuk memprediksi perilaku sirkuit digital dan HDL. Setelah simulasi berhasil, penulis menggunakan *compiler* dari *vendor* Altera untuk melakukan *technology mapping*, *placement & routing*, dan *bitstream generation*. Jika sudah berhasil, *bitstream* dapat diunggah ke FPGA aktual.

3.2 Desain Sistem

Diagram *Top-Level* dari *processor core* yang diajukan terdapat dalam Gambar 3.2. Dalam desain ini, prosesor dibagi menjadi lima tahap: *Instruction Fetch*, *Instruction Decode*, *Execution*, *Memory*, dan akhirnya *Write Back*. Kelima tahap tersebut masing-masing dipisahkan oleh sebuah *pipeline register*. Desain dengan *pipeline* bertujuan untuk



Gambar 3.2: Diagram *top-level* dari *core* prosesor.

mengurangi panjang dari *critical path* yang mempengaruhi frekuensi maksimal dari suatu desain sehingga *throughput* juga dapat ditingkatkan. Sebuah instruksi berjalan sepenuhnya jika sudah melalui kelima tahap tersebut. Dalam setiap *clock*, instruksi baru masuk ke tahap *Instruction Fetch* dan instruksi sebelumnya masuk ke tahap *Instruction Decode*, dan seterusnya. Ini memungkinkan sejumlah instruksi dijalankan bersamaan, tidak seperti pada prosesor dengan desain *single-cycle*, dimana satu instruksi harus menyelesaikan semua tahap terlebih dahulu sebelum instruksi baru bisa dieksekusi.

Dalam tahap pertama, yakni tahap *Instruction Fetch*, tedapat beberapa komponen fungsional *Program Counter* dan *Program Memory*. *Program Counter* merupakan suatu register yang mengeluarkan *input* dari *clock* sebelumnya ke *output* pada saat *rising edge clock* berikutnya. *Program Counter* bertugas memberikan alamat kepada *Program Memory* supaya alamat memori dan urutan penjalanan program tepat. Sedangkan, *Program Memory* merupakan tempat penyimpanan program yang dieksekusi dalam tahap-tahap berikutnya. Memori ini didesain sebagai suatu *Read-Only Memory* yang tidak dapat diubah selain pada saat pemrograman. Nilai yang disimpan dalam *Program Memory* adalah nilai biner dari program RISC-V *Assembly* yang sudah di-*disassemble* menggunakan *compiler*. Nilai dari *Program Counter* pada *clock* berikutnya ditentukan oleh sekumpulan multiplekser yang ditentukan oleh suatu sinyal kontrol dari tahap-tahap lainnya.

Selanjutnya, pada tahap *Instruction Decode*, memiliki tujuan untuk mendekodekan instruksi dan menyiapkan sinyal kontrol sesuai dengan tipe instruksi tersebut. Tahap ini memiliki sejumlah komponen, diantaranya: *immediate generator*, *register files*, *ALU Controller*, *Control Unit*, serta *Hazard Detection Unit*. *Immediate generator* berfungsi sebagai penentu nilai *immediate* yang dibutuhkan sejumlah instruksi bertipe *I-type*. *Register files* merupakan tempat penyimpanan *user-visible register* dari RISC-V *ISA*, dari *register x0* hingga *register x31*. Sedangkan, *Control Unit* merupakan otak dari arsitektur prosesor ini, dimana *Control Unit* menentukan sinyal kontrol yang tepat sesuai dengan tipe suatu instruksi. Sinyal kontrol ini dilanjutkan ke tahap-tahap berikutnya melalui *pipeline register*. Selain itu, terdapat *ALU controller* yang mengirimkan sinyal kontrol ke *ALU* pada tahap *Excecution* sesuai dengan tipe komputasi yang diperlukan. Terakhir, terdapat *Hazard Detection Unit* yang digunakan untuk memperbaiki kesalahan yang terjadi akibat

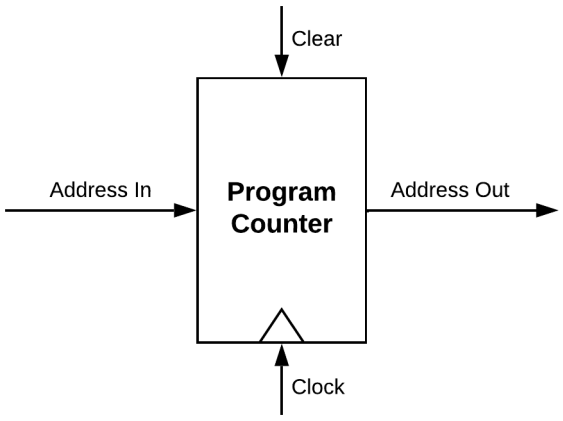
data hazard pada instruksi bertipe *Load*.

Tahap *Execution* memiliki fungsi untuk menjalankan operasi komputasi yang diperlukan sesuai dengan tipe instruksi dari tahap-tahap sebelumnya. Selain itu, pada tahap ini juga diselesaikan permasalahan *hazard* yang timbul karena proses *pipelining*. Dalam tahap ini, terdapat unit fungsional berupa *Arithmetic Logic Unit (ALU)*, *Flushing Unit*, *Forwarding Unit*, *Jump Target Unit*, dan lainnya. *Forwarding Unit* bertujuan untuk menentukan *operand* yang tepat untuk dijalankan oleh *ALU*. *Operand* ini dapat berasal dari *pipeline register ID/EX* atau dari tahap-tahap berikutnya. *Jump Target Unit* bertugas untuk mencegah data yang salah masuk ke dalam *pipeline register* IF/ID dan ID/EX pada *clock* berikutnya. Sedangkan, *Jump Target Unit* berfungsi untuk meneruskan hasil kalkulasi dalam *ALU* ke multiplekser *Program Counter* pada saat tipe instruksi *Jump* atau *Branch*.

Pada tahap *Memory*, data dan sinyal dari tahap sebelumnya diproses oleh *Data Memory*. *Data Memory*, bekerja seperti sebuah *Random-Access Memory* dimana ia dapat ditulis atau dibaca sesuai dengan *input* secara bebas. Selain proses pembacaan dan penulisan memori, tahap ini juga menerukan sejumlah sinyal dari tahap sebelumnya ke tahap *Write Back*. Sinyal tersebut diantanya: sinyal kontrol, alamat *write address register files*, hasil kalkulasi *ALU* dari tahap sebelumnya, dan sebagainya.

Terakhir, pada tahap *Write Back*, diteruskan nilai-nilai dari tahap sebelumnya ke *Register Files*. Terdapat sebuah multiplekser yang memilih *output* dari *input* yang merupakan nilai dari hasil kalkulasi *ALU* dan data yang dikeluarkan oleh *Data Memory* dari tahap-tahap sebelumnya. Nilai tersebut diteruskan ke *input Register Files* sesuai dengan tipe instruksi. Juga terdapat sinyal *write enable* dan *write address* yang ditujukan ke *Register Files*.

3.2.1 *Program Counter*



Gambar 3.3: Diagram Blok *Program Counter*.

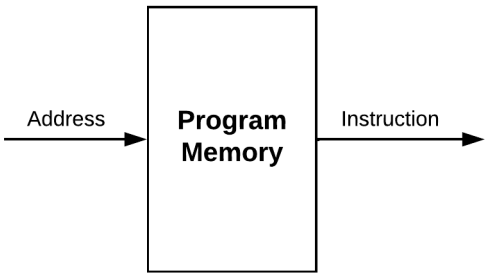
Program counter merupakan register yang menyimpan nilai alamat *program memory* yang digunakan pada *clock* berikutnya. Nilai yang masuk ke dalam *program counter* adalah nilai yang dipilih oleh multiplekser masukan *program counter* dengan sinyal kontrol yang relevan. Dalam arsitektur RISC-V, terdapat sejumlah instruksi yang nilainya

Tabel 3.1: Deskripsi *input/output* dari *Program Counter*

Nama	**Tipe**	**Panjang *Bus***	**Deskripsi**
clk	*input*	1 bit	sinyal *clock* dari sistem, *positive edge triggered*
clr	*input*	1 bit	*asynchronous* clear
address_in	*output*	32 bit	*address* masukan dari multiplekser *Program Counter*
address_out	*output*	32 bit	*address* keluaran *Program Counter* pada *clock* berikutnya

relatif terhadap *program counter*. Sehingga, nilai dari *program counter* bisa digunakan untuk kalkulasi ke dalam *ALU* dengan nilai *immediate*, seperti pada instruksi AUIPC. Gambar 3.3 merupakan diagram blok dari *Program Counter* yang *positive-edge triggered*. Sedangkan Tabel 3.1 merupakan deskripsi dari masukan dan keluaran *program counter* yang digunakan.

3.2.2 *Program Memory*



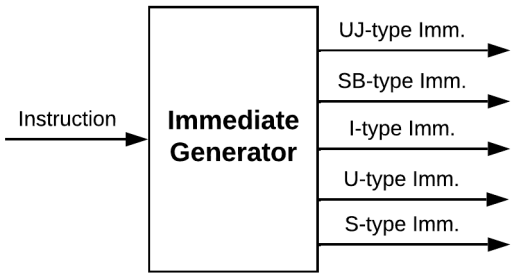
Gambar 3.4: Diagram Blok *Program memory*.

Tabel 3.2: Deskripsi *input/output* dari *Program Memory*

Nama	**Tipe**	**Panjang *Bus***	**Deskripsi**
clk	*input*	1 bit	sinyal *clock* dari sistem, *positive edge triggered*
address	*input*	32 bit	Panjang maksimal 32 bit
instruction_out	*output*	32 bit	Merupakan data yang tersimpan dalam suatu *memory address*

Program memory, atau yang biasa dikenal sebagai *instruction memory*, merupakan memori yang menyimpan semua instruksi program yang dijalankan. Semua instruksi dimulai dengan *program counter* yang memberikan suplai alamat ke dalam *program memory*. Keluaran dari *program memory* digunakan dalam *datapath* sesuai dengan tipe instruksi yang dapat dideteksi dari *OPcode*. *Program memory* diimplementasikan sebagai sebuah *Read-Only Memory* yang bersifat kombinasional. Diagram blok dari *program memory* terdapat pada Gambar 3.17. Sedangkan, penjelasan masukan dan keluaran dari *program memory* terdapat dalam Tabel 3.11.

3.2.3 Immediate Generator



Gambar 3.5: Diagram Blok *Immediate Generator*.

Tabel 3.3: Deskripsi *Input/Output* dari *Immediate Generator*

Nama	**Tipe**	**Panjang *Bus***	**Deskripsi**
instruction	*input*	1 bit	Instruksi dari *pipeline register IF/ID*
s_type_imm	*output*	32 bit	immediate untuk *instruction type store*
sb_type_imm	*output*	32 bit	immediate untuk *instruction type branch*
u_type_imm	*output*	32 bit	immediate untuk *instruction Load Upper Immediate*
uj_type_imm	*output*	32 bit	immediate untuk *instruction type* jump and link
i_type_imm	*output*	32 bit	immediate untuk *instruction type register-immediate*

Immediate Generator digunakan untuk menghasilkan nilai *immediate* yang dibutuhkan oleh sejumlah instruksi. Terdapat beberapa tipe *immediate* yang dihasilkan oleh *immediate generator*, yakni: tipe UJ, tipe SB, tipe I, tipe U, dan tipe S. Nilai *immediate* tersebut kemudian dipilih oleh sejumlah multiplekser yang dikendalikan sinyal kontrol dari *control unit*. Kemudian, *immediate* yang dipilih diperlakukan sebagai *offset* yang masuk ke dalam *ALU*. Pengkodean dari *immediate* juga berbeda-beda sesuai dengan tipe instruksi. Ini dapat dilihat pada Gambar 2.15 dan Gambar 2.16. Secara garis besar, *immediate* yang dihasilkan *immediate generator* yakni sebagai berikut:

1. UJ-type merupakan sinyal *immediate* yang digunakan pada tipe instruksi *unconditional jump*. Contoh instruksi yang menggunakan *immediate* ini yakni JAL. `UJ_type_imm = inst(31 downto 25) & inst(11 downto 7)` yang kemudian di-*sign-extend* ke bilangan *signed* 32-bit.

2. SB-type merupakan sinyal *immediate* yang digunakan oleh instruksi *branching*, yang merupakan *conditional jump*. Instruksi yang dimaksud yakni BEQ, BNE, BGE, dan lainnya. `SB_type_imm = inst(31)&inst(7)&inst(30 downto 25)&inst(11 downto 8)&'0'` yang kemudian di-*sign-extend* ke bilangan *signed* 32-bit.

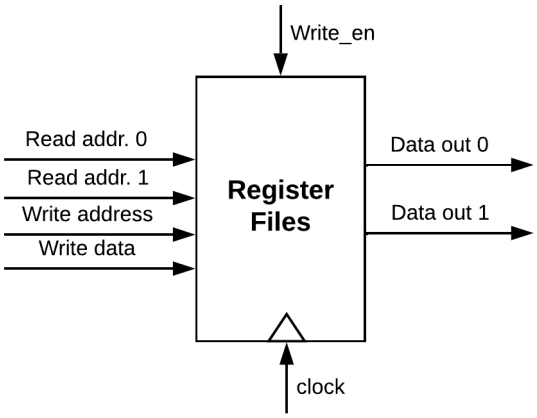
3. I-type digunakan oleh instruksi bertipe I. Instruksi yang menggunakan *immediate* ini contohnya ADDI, XORI, ANDI, dan lainnya. `I_type_imm = inst(31 downto 20)` yang kemudian di-*sign-extend* ke bilangan *signed* 32-bit.

4. U-type digunakan oleh instruksi bertipe *upper-immediate*. Contoh instruksi yang menggunakan *immediate* ini yakni AUIPC (*add upper-immediate to PC*) dan LUI

(*load upper-immediate*). `U_type_imm = inst(31)&inst(30 downto 20)&inst(19 downto 12))&x"000"` yang kemudian di-*sign-extend* ke bilangan *signed* 32-bit.

5. S-type: merupakan sinyal *immediate* yang digunakan pada tipe instruksi *Store*. `S_type_imm = inst(31 downto 25)&inst(11 downto 7)` yang kemudian di-*sign-extend* ke bilangan *signed* 32-bit.

3.2.4 Register Files



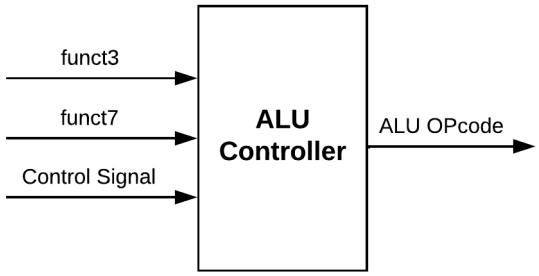
Gambar 3.6: Diagram Blok *Register Files*.

Tabel 3.4: Deskripsi *Input/Output* dari *Register Files*

Nama	**Tipe**	**Panjang *Bus***	**Deskripsi**
clk	*input*	1 bit	sinyal clock dari sistem, *positive edge triggered*
write_en	*input*	1 bit	Menentukan apakah *register* ditulis atau tidak
read_address_0	*input*	5 bit	Memilih *register* yang dikeluarkan pada *output0*
read_address_1	*input*	5 bit	Memilih *register* yang dikeluarkan pada *output* 1
write_address	*input*	5 bit	Merupakan alamat yang dituliskan saat write_en aktif
write_data	*input*	32 bit	merupakan data yang dituliskan ke write_address saat write_en aktif
data_out_0	*output*	32 bit	Mengeluarkan data dengan alamat *register* read_address_0
data_out_0	*output*	32 bit	Mengeluarkan data dengan alamat *register* read_address_1

Register Files merupakan unit fungsional yang menyimpan 32 register *user-visible* dalam arsitektur ISA RISC-V. Nilai dari register `x0` selalu bernilai 0 yang dapat dimanfaatkan untuk sejumlah *pseudoinstruction*. Unit ini dikendalikan oleh sinyal kontrol dari instruksi dan *control unit*. *Register files* mengeluarkan pembacaan dua *data* sekaligus dari alamat `rs1` dan `rs2`. Dalam prosesor ini, digunakan *register files* yang *postive-edge triggered*. Sedangkan, proses penulisan ke register dikendalikan oleh sinyal masukan *write destination* `rd`, WriteData dan WriteEn. Gambar 3.6 merupakan diagram blok dari *Register Files* yang digunakan. Sedangkan, Tabel 3.4 merupakan deskripsi dari masukan dan keluaran *Register Files*.

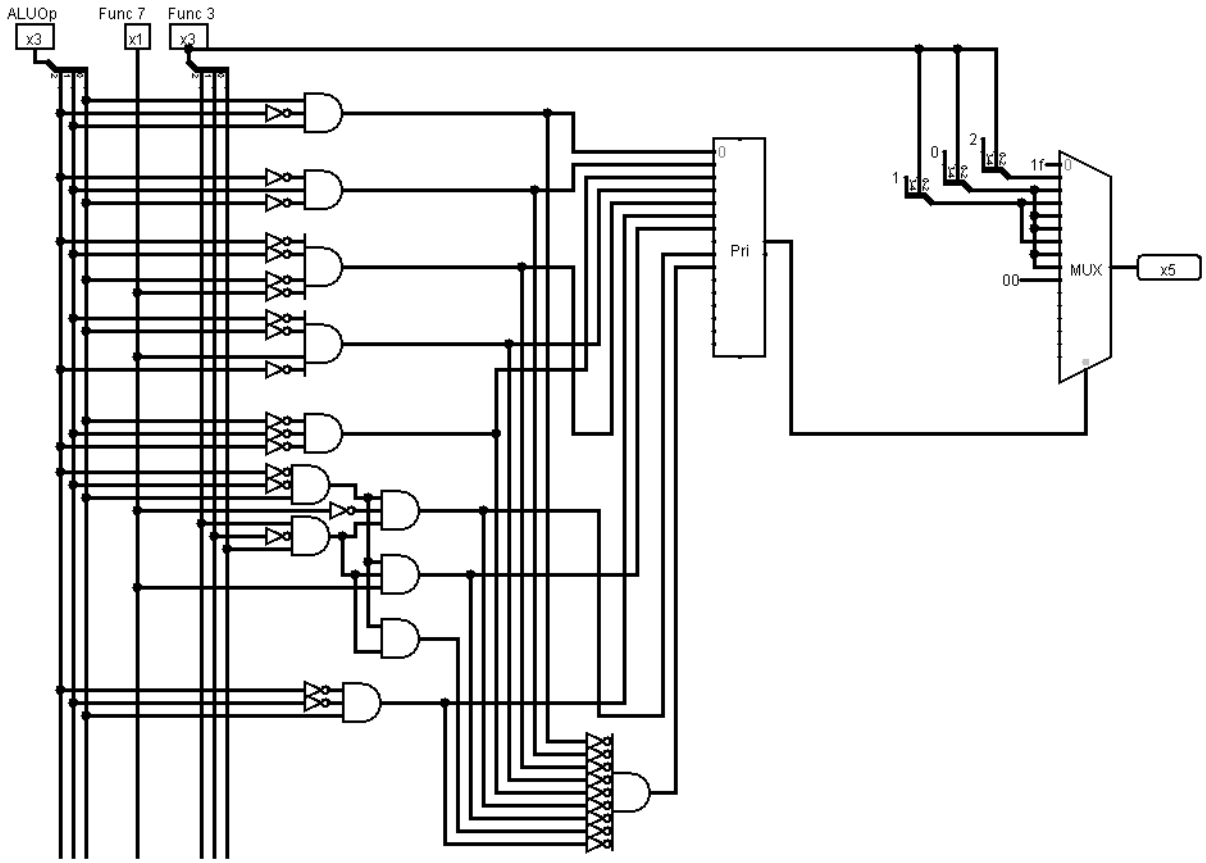
3.2.5 ALU Controller



Gambar 3.7: Diagram Blok *ALU Controller*.

Tabel 3.5: Deskripsi *Input/Output* dari *ALU Controller*.

Nama	**Tipe**	**Panjang Bus**	**Deskripsi**
ALU_control	*input*	1 bit	sinyal kendali yang berasal dari control unit
func7	*input*	1 bit	menentukan jenis instruksi, berasal dari instruction
func3	*input*	3 bit	menentukan jenis instruksi, berasal dari instruction
ALU Opcode	*output*	5 bit	menentukan operasi ALU sesuai tipe instruksi

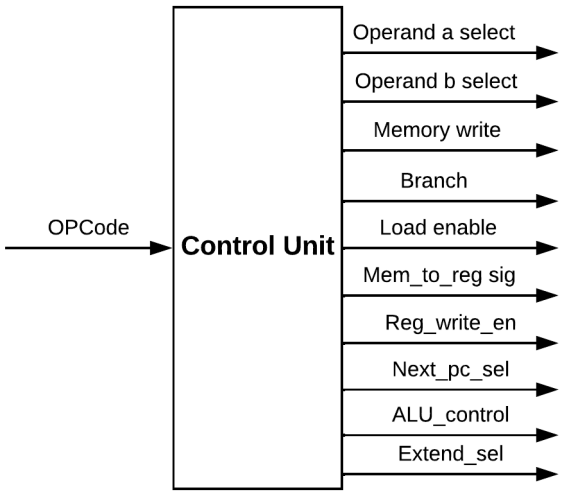


Gambar 3.8: Implementasi *Gate-level* dari *ALU Controller* (Rafique, 2019)

ALU Controller digunakan untuk menghasilkan sinyal *ALU OP Code* yang digunakan oleh *ALU* untuk memproses *operand* masukan sesuai dengan tipe instruksi. Sinyal masukan ke *ALU Controller* yakni sinyal dari *control unit*, funct7, dan funct3 yang didapatkan dari instruksi. Gambar 3.7 merupakan diagram blok dari *ALU Controller* dan Tabel 3.5 merupakan deskripsi dari masukan dan keluarannya. Secara *Gate-level*, *ALU Controller* diterapkan sebagai sebuah *priority encoder*. Ini dapat dilihat dalam Gambar 3.8 Berikut ini merupakan deskripsi penentuan nilai *ALU OPcode*, dari prioritas tertinggi ke yang terendah:

1. `ALU_OP = '1'&x"f"` pada saat `ALU_control = "011"`
2. `ALU_OP = "10"&funct3` pada saat `ALU_control = "010"`
3. `ALU_OP = "00"&funct3` pada saat `ALU_control = "000"` dan `funct7 = '0'`
4. `ALU_OP = "01"&funct3` pada saat `ALU_control = "000"` dan `funct7 = '1'`
5. `ALU_OP = "00"&funct3` pada saat `ALU_control = "000"`
6. `ALU_OP = "00"&funct3` pada saat `ALU_control = "001"` dan `funct7 = '0'` dan `funct3 = '101'`
7. `ALU_OP = "01"&funct3` pada saat `ALU_control = "001"` dan `funct7 = '1'` dan `funct3 = '101'`
8. `ALU_OP = "00"&funct3` pada saat `ALU_control = "001"` dan `funct3 = '101'`
9. `ALU_OP = "00"&funct3` pada saat `ALU_control = "001"`
10. `ALU_OP = "00000"` selain dari syarat-syarat sebelumnya.

3.2.6 Control Unit



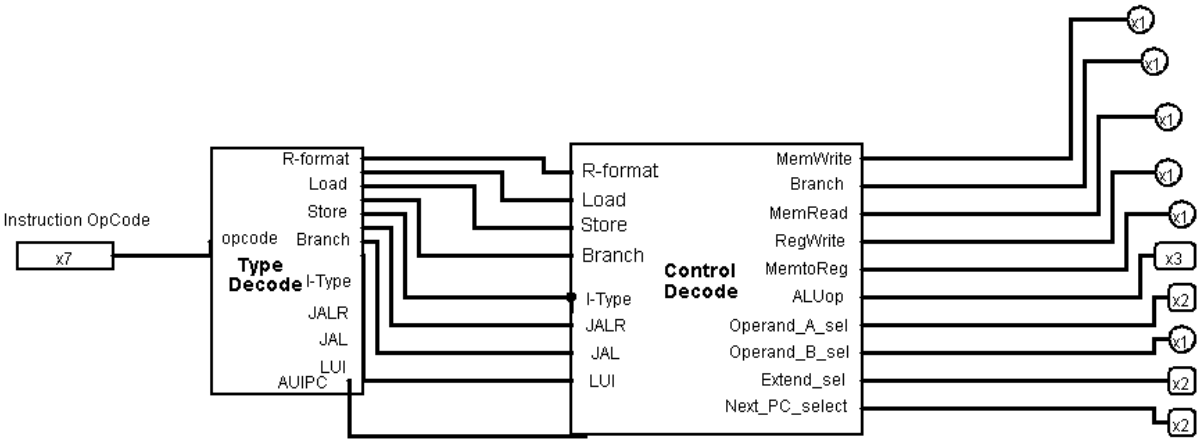
Gambar 3.9: Diagram Blok *Control Unit*.

Tabel 3.6: Deskripsi *input/output* dari *Control Unit*

Nama	**Tipe**	**Panjang *Bus***	**Deskripsi**
opcode	**input**	7 bit	Merupakan informasi tipe operasi dari instruksi
mem_write	**output**	1 bit	Sinyal kontrol *data memory*
branch	**output**	1 bit	Sinyal kontrol tipe instruksi *branch*
reg_write	**output**	1 bit	sinyal kontrol penulisan *register files*
mem_to_reg	**output**	1 bit	sinyal kontrol pemilihan data penulisan *register files*
load_en	**output**	1 bit	sinyal kontrol untuk tipe instruksi *load*
operand_a_sel	**output**	2 bit	memilih operand a yang masuk *ALU*
operand_b_sel	**output**	1 bit	memilih operand b yang masuk ke *ALU*
next_pc_sel	**output**	2 bit	sinyal kontrol multiplekser *program counter*
extend_sel	**output**	2 bit	memilih *output immediate generator* sesuai tipe instruksi
ALU_OP	**output**	3 bit	sinyal kontrol *ALU Controller*

Tabel 3.7: *Truth Table* parsial dari unit kontrol.

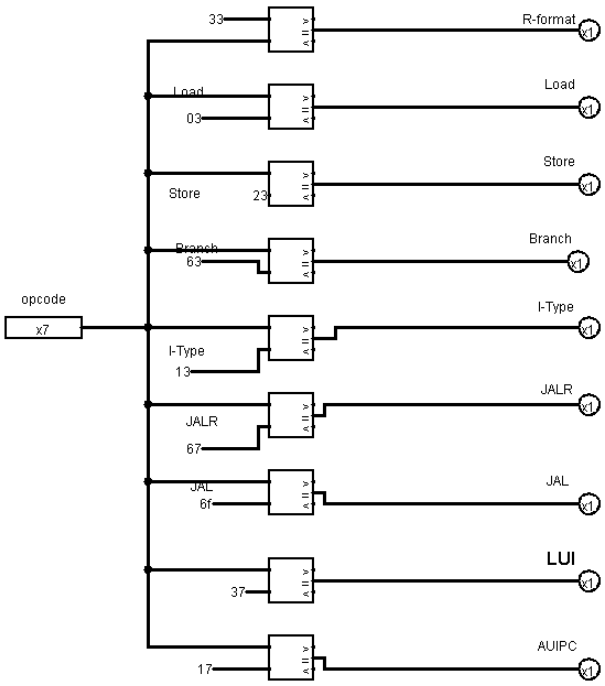
Tipe Instruksi	**Sinyal Kontrol**				
	MemWrite	**Branch**	**RegWrite**	**MemtoReg**	**operand_B_sel**
R-Type	0	0	1	0	0
Load	0	0	1	1	1
Store	1	0	0	0	1
Branch	0	1	0	0	0
I-Type	0	0	1	0	1
JALR	0	0	1	0	0
JAL	0	0	1	0	0
LUI	0	0	1	0	1
AUIPC	0	0	1	0	1



Gambar 3.10: *Control Unit* dapat dilihat sebagai satu kesatuan dari tahap *type-decode* dan *control-decode*(Rafique, 2019).

Control Unit merupakan otak dari prosesor yang mengendalikan keseluruhan unit fungsional dalam arsitektur sesuai dengan tipe instruksi. Unit ini dapat dibagi menjadi dua tahap yakni tahap *type-decode* dan tahap *control-decode*. Kedua tahap ini dapat dilihat dalam Tabel 3.11 dan Tabel 3.12. Tabel 3.7 merupakan *truth table* dari *control unit* prosesor. Sinyal kontrol dari *control unit* digunakan untuk mengendalikan komponen pada arsitektur prosesor sesuai dengan tipe instruksi. Tipe instruksi yang digunakan untuk proses *control decoding* didapatkan dari *OP code* yang merupakan 7 bit terendah dari instrruksi. Untuk menentukan tipe dari instruksi, digunakan konvensi seperi Gambar 2.3, sebagai berikut:

1. Tipe R jika `OPCode = x"33"`
2. Tipe *Load* jika `OPCode = x"03"`
3. Tipe *Store* jika `OPCode = x"23"`
4. Tipe *Branch* jika `OPCode = x"63"`
5. Tipe I jika `OPCode = x"13"`

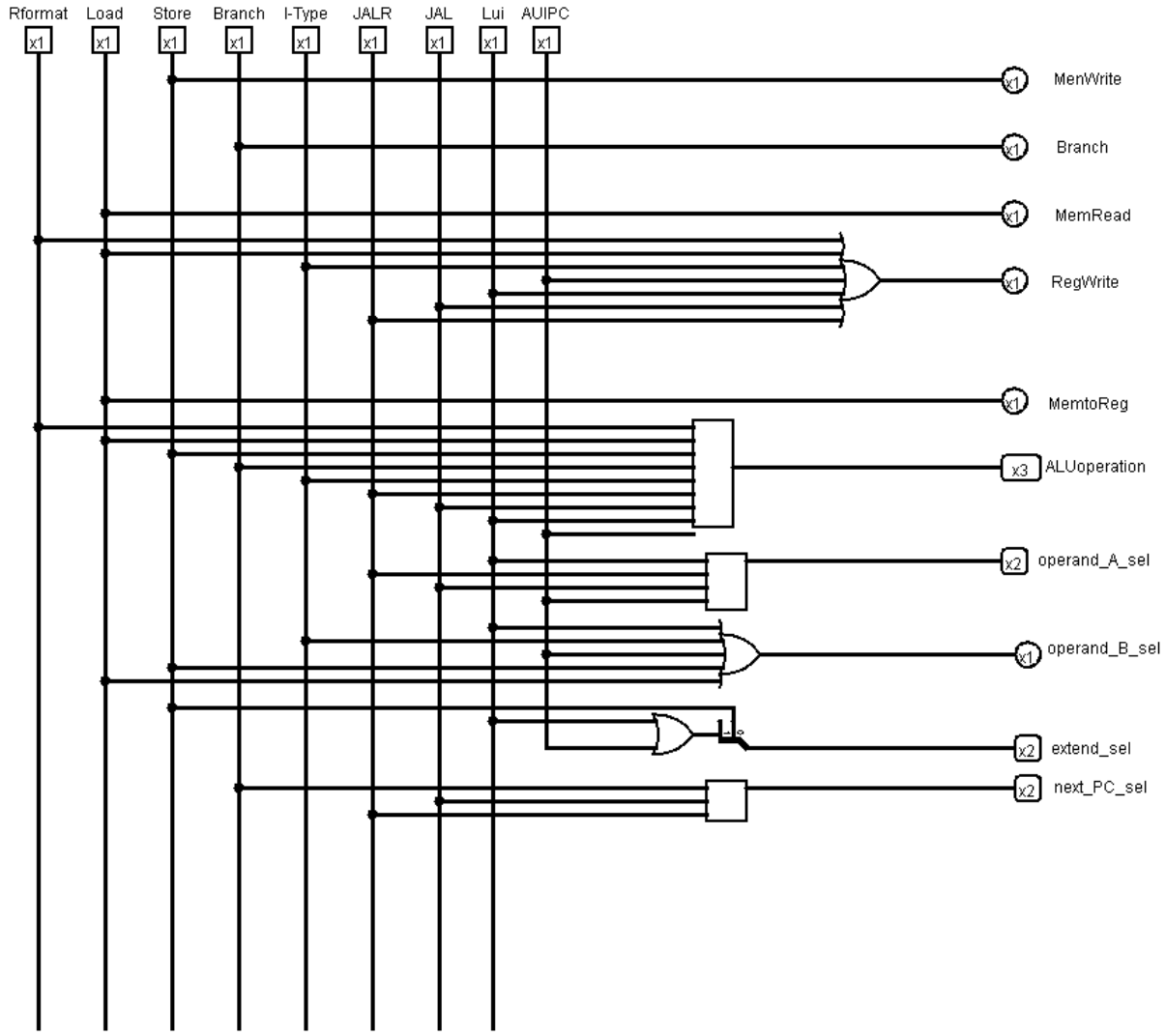


Gambar 3.11: Implementasi tahap *type-decode* dari *Control Unit*(Rafique, 2019).

6. Tipe JALR jika `OPCode = x"67"`
7. Tipe JAL jika `OPCode = x"6f"`
8. Tipe LUI jika `OPCode = x"37"`
9. Tipe AUIPC jika `OPCode = x"17"`

Sedangkan, penjelasan dari setiap sinyal kontrol yakni sebagai berikut:

1. MemWrite merupakan sinyal kontrol yang digunakan untuk mengendalikan *data memory*. Pada saat `MemWrite = 1`, maka *data memory* menuliskan *input* ke dalam alamat memori yang menjadi masukan. Sedangkan, pada saat `MemWrite = 0`, *data memory* tidak menuliskan *input* ke dalam register. Pada RISC-V hanya instruksi *Load* dan *Store* saja lah yang membutuhkan akses memori. Ini tercerminkan dalam logika MemWrite yang hanya aktif pada saat tipe instruksi *Store*.
2. Branch merupakan sinyal kontrol yang digunakan untuk menandakan instruksi *branching* ke *ALU Controller*. Sinyal kontrol ini aktif pada saat *OP Code* menandakan tipe instruksi *branch*. Pada saat sinyal kontrol `Branch = 1`, maka nilai keluaran *ALU* pada operasi *branching* digunakan untuk menentukan keluaran multiplekser ke *Program Counter*.
3. RegWrite digunakan untuk mengendalikan penulisan *Register Files*. Penulisan pada *Register Files* sesuai dengan alamat masukan aktif pada saat `RegWrite = 1` dan sebaliknya pada saat `RegWrite = 0`. Pada RISC-V, akses *Register Files* dilakukan oleh semua tipe inti kecuali tipe *Load* dan *Store*. Destinasi dari penulisan yakni *register destination* `rd`.



Gambar 3.12: Implementasi tahap *control-decode* dari *Control Unit*(Rafique, 2019).

4. MemtoReg pada *control unit* berperan untuk memilih keluaran multiplekser pada tahap EX/MEM. Multiplekser mengeluarkan masukan dari *ALU* pada saat `MemtoReg = 0`. Sebaliknya, multiplekser mengeluarkan masukan dari *data memory* pada saat `MemtoReg = 1`. MemtoReg sendiri aktif pada saat tipe instruksi yang terdeteksi bertipe *Load*. Ini karena pada *ISA* RISC-V, hanya instruksi *Load* saja lah yang memerlukan masukan dari *data memory* ke *register files*.

5. LoadEn merupakan sinyal yang digunakan untuk memilih *Byte Mode* dari proses *Load*. *Byte Mode* merupakan tipe penulisan data ke *data memory*. Terdapat sejumlah mode, diantaranya: *word*, *halfword*, dan *byte*.

6. OperandASel memilih nilai yang masuk ke dalam masukan *operand* pertama pada *ALU*. OperandASel diterapkan sebagai logika pengambilan keputusan berikut:

   (a) `OperandASel="11"` ketika `lui_type = '1'`

   (b) `OperandASel="10"` ketika `jalr_type = '1'`

   (c) `OperandASel="10"` ketika `jal_type = '1'`

(d) `OperandASel="01"` ketika `auipc_type = '1'`

(e) `OperandASel="00"` ketika selain dari syarat-syarat sebelumnya.

7. OperandBSel memilih nilai yang masuk ke dalam masukan *operand* kedua pada *ALU*. `OperandBSel = 1` pada saat nilai *operand* kedua dalam *ALU* berasal dari *immediate generator*. Tipe instruksi yang menggunakan nilai *immediate*, yakni: *Load*, *Store*, Tipe I, *LUI*, dan *AUIPC*.

8. NextPCSel memilih nilai yang menjadi nilai berikutnya pada *program counter*. NextPCSel diterapkan sebagai logika pengambilan keputusan berikut:

   (a) `NextPCSel="01"` ketika `branch_type = '1'`

   (b) `NextPCSel="10"` ketika `jal_type = '1'`

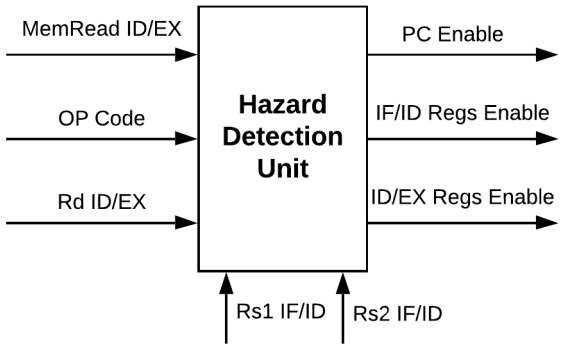
   (c) `NextPCSel="11"` ketika `jalr_type = '1'`

   (d) `NextPCSel="00"` ketika selain dari syarat-syarat sebelumnya.

9. ExtendSel memilih nilai *immediate* yang sesuai dengan tipe instruksi. ExtendSel diterapkan sebagai logika `ExtendSel = store_type&(lui_type or auipc_type)`.

10. ALUOp digunakan oleh *ALU Controller* untuk memilih *ALU OP Code* yang sesuai dengan tipe instruksi serta `funct7` dan `funct3`. ALUOp diterapkan sebagai logika `ALUOp = not (r_type or branch_type or i_type or jal_type or jalr_type) & not(r_type or load_type or store_type or i_type) & not(r_type or load_type or branch_type or lui_type or auipc_type)`.

3.2.7 *Hazard Detection Unit*



Gambar 3.13: Diagram Blok *Hazard Detection Unit*.

Gambar 3.13 merupakan blok diagram dari *Hazard Detection Unit*. *Hazard Detection Unit* merupakan unit fungsional yang berguna untuk memperbaiki kesalahan yang terjadi akibat *Data Hazard* yang berkaitan dengan instruksi *Load*. Pada saat instruksi *Load*, misalnya `lw`, `lh`, dan `lb` dijalankan dalam tahap *Execution*, dapat terjadi *Data Hazard*. *Hazard* ini terjadi karena instruksi berikutnya yang berada di tahap *pipeline* sebelum *Execution* membutuhkan data yang masih di proses oleh instruksi *Load*. *Forwarding Unit* mampu mengatasi *Data Hazard* akibat instruksi lain selain *Load*. *Data Memory* yang

Tabel 3.8: Deskripsi input/output dari Hazard Detection Unit.

Nama	**Tipe**	**Panjang *Bus***	**Deskripsi**
opcode	input	7 bit	Merupakan informasi tipe operasi dari instruksi
memread_IDEX	input	1 bit	sinyal kendali penulisan register files pada tahap ID/EX
rs1_IFID	input	5 bit	alamat asal pembacaan keluaran pertama register files
rs2_IFID	input	5 bit	alamat asal pembacaan keluaran kedua register files
rd_IDEX	input	5 bit	alamat tujuan penulisan register files pada tahap EX
pc_en	output	1 bit	sinyal kendali aktivasi program counter
regs_en_IFID	output	1 bit	sinyal kendali aktivasi pipeline register tahap IF/ID
ctrl_mux_en	output	1 bit	sinyal kendali aktivasi kontrol unit dan register ID/EX

digunakan dalam rancangan bersifat aktif *low*, yang mengakibatkan *data* nilai register berikutnya masih dalam proses pada saat *rising edge clock*. Akibatnya, *Forwarding Unit* memerlukan *stall* selama satu periode *clock* yang diimplementasikan sebagai `NOP`. Tugas dari *Hazard Detection Unit* yakni untuk mendeteksi kapan diperlukannya *stall* dan juga mengeluarkan sinyal *stall* pada saat *data hazard* terdeteksi pada tahap *Execution*.

Tabel 3.8 menjelaskan keluaran dan masukan *Hazard Detection Unit* beserta dengan deskripsi singkatnya. *Hazard Detection Unit* yang dirancang memiliki lima *input* dan tiga *output*. *OP Code* merupakan 7 bit pertama dari instruksi pada tahap *Instruction Decode*. Tipe instruksi yang dideteksi kemudian digunakan untuk pemilihan keputusan mengenai proses *stall*. Sinyal `memread_IDEX` merupakan penanda instruksi bertipe *Load* pada tahap *Execution*. Pada saat `memread_IDEX` aktif, instruksi yang berada dalam tahap *Execution* bertipe *Load* yang berpotensi mengakibatkan *data hazard*. Selanjutnya, `rs1_IFID`, `rs2_IFID`, dan `rd_IDEX` merupakan alamat register asal pembacaan dalam tahap *Instruction Decode* dan alamat register tujuan penulisan pada tahap *Execution*. Sedangkan, `pc_en`, `regs_en_IFID`, dan `ctrl_mux_en` merupakan sinyal kendali yang mengaktifkan atau menonaktifkan *program counter*, *pipeline register IF/ID*, dan unit kontrol. Prosesor berada dalam kondisi *stall* pada saat ketiga sinyal keluaran tersebut bersifat nonaktif.

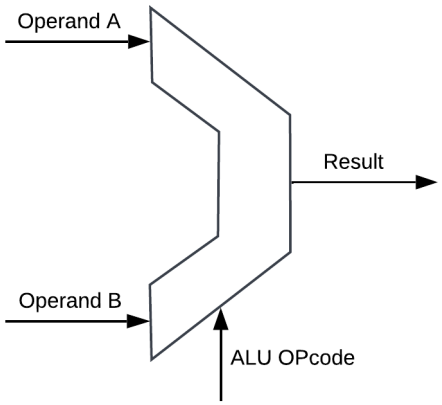
Perilaku dari *Hazard Detection Unit* dapat dijelaskan sebagai berikut:

1. Pada saat `memread_IDEX = '1' and (opcode = "0110111" or opcode = "0100011" or opcode="0000011" or opcode="0010011" or opcode ="0010111") and (rs1_IFID = rd_IDEX)`,
   `regs_en_IFID <= '0'`,
   `pc_en <= '0'`, dan
   `ctrl_mux_en <= '0'`

2. Pada saat `memread_IDEX = '1' and (rs1_IFID = rd_IDEX or rs2_IFID = rd_IDEX)`
   `regs_en_IFID <= '0'`,
   `pc_en <= '0'`, dan
   `ctrl_mux_en <= '0'`

3. Selain dari syarat-syarat sebelumnya:
   `regs_en_IFID <= '1'`,
   `pc_en <= '1'`, dan
   `ctrl_mux_en <= '1'`

3.2.8 *Arithmetic Logical Unit*

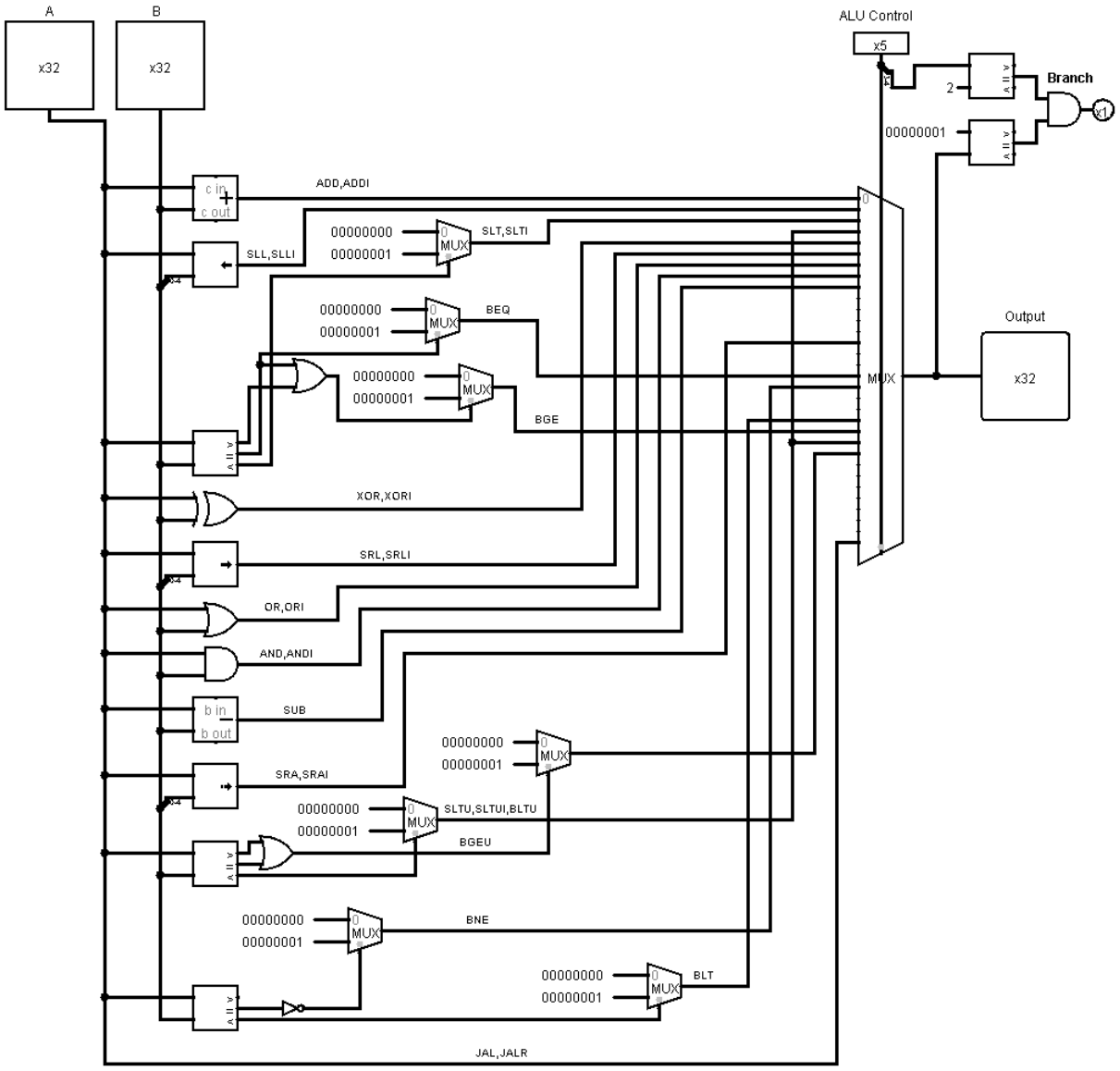


Gambar 3.14: Diagram Blok *Arithmetic Logic Unit*.

Tabel 3.9: Deskripsi *input*/*output* dari ALU

Nama	**Tipe**	**Panjang *Bus***	**Deskripsi**
ALUop_in	*input*	5 bit	sinyal kendali *ALU* dari *ALU Controller*
operand_a	*input*	32 bit	*Operand* pertama untuk diproses
operand_b	*input*	32 bit	*Operand* kedua untuk diproses
result_out	*output*	32 bit	Hasil dari pemrosesan kedua *operand*

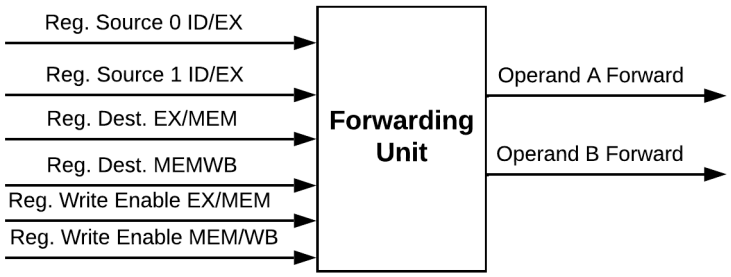
1. Pada saat `ALUControl = "00000"`, `ALUOut = A + B`, *unsigned*.
2. Pada saat `ALUControl = "00001"`, `ALUOut = A shift left B(4 downto 0)`, *unsigned*
3. Pada saat `ALUControl = "00010"`, `ALUOut = 1 when A < B, else 0`, *signed*
4. Pada saat `ALUControl = "00011"`, `ALUOut = 1 when A < B, else 0`, *unsigned*
5. Pada saat `ALUControl = "00100"`, `ALUOut = A XOR B`
6. Pada saat `ALUControl = "00101",ALUOut = A shift right B(4 downto 0)`, *unsigned*
7. Pada saat `ALUControl = "00110"`, `ALUOut = A OR B`
8. Pada saat `ALUControl = "00111"`, `ALUOut = A AND B`
9. Pada saat `ALUControl = "01000"`, `ALUOut = A - B`, *unsigned*
10. Pada saat `ALUControl = "01101"`, `ALUOut = A shift right B(4 downto 0)`, *signed*
11. Pada saat `ALUControl = "10000"`, `ALUOut = 1 when A = B, else 0`, *signed*



Gambar 3.15: Implementasi *Gate-level* dari *Arithmetic Logic Unit* (Rafique, 2019).

12. Pada saat `ALUControl = "10001"`, `ALUOut = 0 when A = B, else 1`, *signed*
13. Pada saat `ALUControl = "10100"`, `ALUOut = 1 when A < B, else 0`, *signed*
14. Pada saat `ALUControl = "10101"`, `ALUOut = (1 when A = B, else 0) OR (1 when A > B else 0)`, *signed*
15. Pada saat `ALUControl = "10110"`, `ALUOut = 1 when A < B, else 0`, *unsigned*
16. Pada saat `ALUControl = "10111"`, `ALUOut = (1 when A = B, else 0) OR (1 when A > B else 0)`, *unsigned*
17. Pada saat `ALUControl = "11111"`, `ALUOut = A`.
18. `ALUControl = "00000"` pada saat NOP atau instruksi *invalid*.

3.2.9 *Forwarding Unit*



Gambar 3.16: Diagram Blok *Forwarding Unit*.

Tabel 3.10: Deskripsi input/output dari *Forwarding Unit*

Nama	**Tipe**	**Panjang *Bus***	**Deskripsi**
rs0_idex	*input*	5 bit	alamat asal data 0 dari *Register Files* pada tahap *EX*
rs1_idex	*input*	5 bit	alamat asal data 1 dari *Register Files* pada tahap *EX*
rd_exmem	*input*	5 bit	alamat penulisan *Register Files* pada tahap *MEM*
rd_memwb	*input*	5 bit	alamat penulisan *Register Files* pada tahap *WB*
regwrite_exmem	*input*	1 bit	sinyal kendali penulisan *Register Files* pada tahap *MEM*
regwrite_memwb	*input*	1 bit	sinyal kendali penulisan *Register Files* pada tahap *WB*
forward_mux_a	*output*	2 bit	sinyal yang memilih keluaran dari multiplekser *ALU* a
forward_mux_b	*output*	2 bit	sinyal yang memilih keluaran dari multiplekser *ALU* b

Forwarding Unit merupakan unit fungsional yang berguna untuk melakukan penanganan *data hazard*. *Data hazard* terjadi pada saat terdapat kebergantungan nilai register yang belum selesai diproses dari *cycle* sebelumnya. *Forwading* bekerja dengan meneruskan nilai yang terdapat dalam satu tahap *pipeline* ke tahap sebelumnya tanpa harus menulis nilai ke register terlebih dahulu. Peristiwa *forwarding* dapat dilihat dalam Gambar 2.19. Untuk mendeteksi kapan perlunya *forwarding* dilakukan, perlu dilakukan perbandingan antara *register source* pada beberapa tahap dalam *pipeline*.*Forwarding unit* menentukan nilai *operand* yang masuk ke dalam *ALU* melalui multiplekser. Berikut ini merupakan algoritma yang digunakan untuk menentukan *data* yang di-*forward*.

Untuk sinyal ForwardMuxA:

1. `ForwardMuxA = "10"` ketika `RegWrite.EXMEM = '1' AND rd.EXMEM != "00000" AND rd.EXMEM = rs1.IDEX`

2. `ForwardMuxA = "01"` ketika `RegWrite.EXMEM = '0' AND rd.MEMWB != "00000" AND rd.MEMWB = rs1.IDEX`

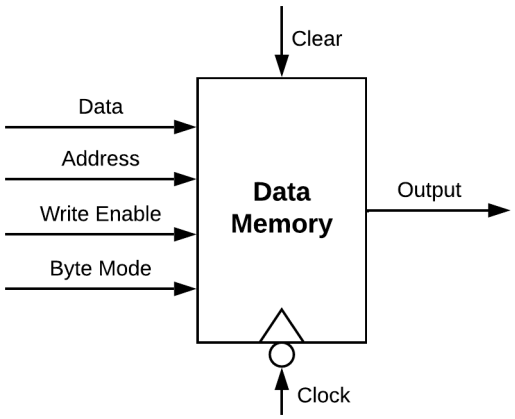
3. `ForwardMuxA = "10"` ketika `RegWrite.MEMWB = '1' AND rd.MEMWB != "00000" AND NOT(RegWrite.EXMEM = '1' AND rd.EXMEM != "00000" AND (rd.EXMEM = rs1.IDEX)) AND rd.MEMWB = rs1.IDEX`

4. `ForwardMuxA = "00"` ketika tidak terjadi *data hazard*.

Sedangkan, untuk sinyal ForwardMuxB:

1. `ForwardMuxB = "10"` ketika `RegWrite.EXMEM = '1' AND rd.EXMEM != "00000" AND rd.EXMEM = rs2.IDEX`
2. `ForwardMuxB = "01"` ketika `RegWrite.EXMEM = '0' AND rd.MEMWB != "00000" AND rd.MEMWB = rs2.IDEX`
3. `ForwardMuxB = "10"` ketika `RegWrite.MEMWB = '1' AND rd.MEMWB != "00000" AND NOT(RegWrite.EXMEM = '1' AND rd.EXMEM != "00000" AND (rd.EXMEM = rs2.IDEX)) AND rd.MEMWB = rs2.IDEX`
4. `ForwardMuxB = "00"` ketika tidak terjadi *data hazard*.

3.2.10 Data Memory



Gambar 3.17: Diagram Blok *Data Memory*.

Tabel 3.11: Deskripsi *input*/*output* dari *Data Memory*

Nama	**Tipe**	**Panjang *Bus***	**Deskripsi**
clk	*input*	1 bit	sinyal *clock* dari sistem, negative edge triggered
data	*input*	32 bit	Data yang dituliskan ke dalam *memory* dengan *address* masukan
address	*input*	32 bit	alamat penulisan dan pembacaan data dari *address* masukan
write_en	*input*	1 bit	merupakan data yang dituliskan ke *address* masukan
byte_mode	*input*	3 bit	menentukan tipe penulisan data
output	*output*	32 bit	merupakan keluaran dari pembacaan memori

Data Memory atau yang biasa dikenal sebagai *Random Access Memory* merupakan memori yang digunakan prosesor untuk menyimpan data sementara. Karena RISC-V merupakan arsitektur *Load* dan *Store*, maka hanya instruksi-instruksi dengan tipe Load dan Store saja lah yang menggunakan *Data Memory*. *Data Memory* yang dirancang dalam arsitektur prosesor ini memiliki sifat aktif *low*. Memori menulis *data input* ke register dengan alamat *address input* sesuai dengan *byte mode* pada saat *falling edge clock* jika *Write Enable* bernilai aktif. Selain itu, memori mengeluarkan nilai register

dengan alamat *input address* ke *output* pada *falling edge clock*. Keseluruhan nilai yang tersimpan dalam *Data Memory* dapat dihapus dengan mengaktifkan masukan *clear*.

Gambar 3.17 merupakan diagram blok dari *Data Memory* dalam rancangan arsitektur prosesor. Memori ini memiliki tujuh masukan dan satu keluaran. Keluaran dari memori, yakni: *clear*, *clock*, *data*, *address*, *write enable*, dan *byte mode*. *Clear* digunakan untuk menghapus semua nilai register dalam memori. Karena sifatnya yang *asynchronous*, memori langsung dihapus nilainya pada saat sinyal *clear* aktif. *Clock* digunakan untuk mengatur *timing* dari memori. Karena memori bersifat aktif *low*, maka proses pembacaan dan penulisan dilakukan pada saat *falling edge* dari *clock* masukan memori. *Data* merupakan nilai dari tahap *pipeline* sebelumnya yang dituliskan pada register memori dengan alamat *address* pada saat *clock* dan *write enable* aktif. Sedangkan, *byte mode* merupakan pengaturan *alignment* dari memori. Pembacaan dan penulisan pada memori yang dirancang mendukung akses *unaligned*. Sedangkan, keluaran dari memori yakni *output* yang merupakan nilai dari register dengan alaman masukan *address*. Deskripsi singkat mengenai keluaran dan masukan *data memory* serta panjang sinyalnya dapat dilihat dalam Tabel 3.11.

[Halaman ini sengaja dikosongkan]

BAB IV
HASIL DAN PEMBAHASAN

Pada bab ini, dilakukan pengujian terhadap *soft processor* RISC-V yang telah dirancang. Hasil yang ingin diketahui yakni penggunaan *resource FPGA*, validitas logika prosesor, dan performa prosesor. Dari hasil pengujian yang didapatkan, digunakan untuk melakukan perbandingan terhadap *soft processor* yang serupa.

4.1 Skenario Pengujian

Pengujian dilakukan dengan melakukan serangkaian simulasi terhadap logika prosesor dan kemudian melakukan pengunggahan *soft processor* ke dalam sebuah *FPGA* aktual. Pertama-tama, dilakukan pengujian sebuah program deret Fibonacci yang dilakukan dalam simulator ModelSim. Setelah *soft processor* berhasil menjalankan program Fibonacci, maka digunakan Quartus Prime untuk mengkompilasi desain *core* dan mengunggahnya ke dalam *FPGA*. Selain itu, dengan Quartus Prime, dilakukan pengujian penggunaan sumber daya, jalur terpanjang desain, frekuensi maksimal, dan lainnya. Setelah diunggah ke *FPGA*, *soft processor* diuji melalui pengamatan fisik. Terakhir, dilakukan pengujian terhadap instruksi lainnya yang terdapat dalam *Instruction Set Architecture* RV32I. Pengujian ini dilakukan di dalam ModelSim, di mana diamati sinyal keluarannya.

4.2 Penggunaan Sumber Daya

Tabel 4.1: Penggunaan sumber daya oleh RISC-V soft processor core yang diajukan.

Revision Name	RISC-V-Pipelined
Top-level Entity Name	top
Family	Cyclone IV E
Device	EP4CE6E22C8
Timing Models	Final
Total logic elements	2,115 / 6,272 (34 %)
Total registers	558
Total pins	16 / 92 (17 %)
Total virtual pins	0
Total memory bits	67,608 / 276,480 (24 %)
Embedded Multiplier	0 / 30 (0 %)
Total PLLs	0 / 2 (0 %)

Tabel 4.1 menjelaskan sumber daya yang dibutuhkan *chipset FPGA* untuk mensintesis *soft processor* hasil rancangan. Informasi ini dihasilkan oleh Quartus Prime *Design Software* pada tahap *Place & Route*. Jumlah sumber daya yang diperlukan dipengaruhi oleh desain arsitektur dan *compiler*. Pada tipe *FPGA* yang digunakan, yakni EP4CE6E22C8, *soft processor* hasil rancangan memerlukan 2.115 *logic elements*, 558 register, dan 67.608 *memory bits*. *Logic elements* merupakan blok-blok dalam *FPGA* yang diatur oleh *compiler* untuk mengubah masukan menjadi keluaran tertentu. *Registers* merupakan jumlah

perangkat keras register yang dibutuhkan oleh *soft processor*. Sedangkan, *memory bits* merupakan jumlah memori yang dibutuhkan oleh unit-unit seperti *Program Memory* dan *Data Memory*.

4.3 Frekuensi Maksimal

Tabel 4.2: Frekuensi maksimal dari hasil sintesis.

Model	***Fmax***	***Restricted Fmax***	***Clock Name***
Slow 1200mV 85C	62.95 MHz	62.95 MHz	clk
Slow 1200mV 0C	68.7 MHz	68.7 MHz	clk

Tabel 4.2 merupakan informasi mengenai frekuensi maksimal yang didukung *soft processor* hasil rancangan. Kompiler dalam Quartus Prime *Design Software* menghasilkan informasi frekuensi maksimal secara otomatis terhadap *clock* dalam rangkaian. *Clock* yang digunakan dalam rangkaian untuk mengatur *soft processor* yakni `clk`. Terdapat sejumlah skenario temperatur papan *FPGA*. Pada saat temperatur *FPGA* 85°C, *soft processor* mampu mendukung frekuensi 62.95 MHz. Sedangkan, *FPGA* dapat mendukung frekuensi yang lebih cepat pada suhu yang lebih rendah, yakni 68.7 MHz pada suhu 0°C.

Tabel 4.3: Komparasi performa antara *processor core* yang diajukan dan beberapa RISC-V *core* lainnya.

Core	***LUT***	***FF***	***MaxFreq***
Core ini	2115	558	62.95
Single-cycle core sebelumnya	2660	1168	46
Maestro	3583	1513	31
PicoRV	1291	568	367
DarkRISCV	1177	246	150
VexRISCV	1993	1345	214
PicoRV32	1291	568	309
SERV	217	174	367
RPU	2943	1103	111
UE RISC-V	3676	2289	124
UE BiRISC-V	15667	6324	87

Tabel 4.3 merupakan perbandingan performa antara *processor core* yang diajukan dengan sejumlah *core* lainnya (Qui et al., 2020). *Soft processor* yang diajukan dapat berjalan dengan kecepatan hingga 62.95 MHz, yang sedikit meningkat dari *soft processor single-cycle* yang dibuat pada penelitian sebelumnya yang berjalan dengan frekuensi maksimal 46 MHz (Aaron Elson, 2022). Dengan metode *pipelining throughput* prosesor meningkat sebesar 37%. Penulis juga membandingkan perfoma dengan *core* Maestro yang menjadi inspirasi dari proposal yang diajukan. Terdapat perkembangan signifikan antara *core* yang diajikan dan *processor core* Maestro. Namun, *core* RISC-V dengan *pipeling* lima tahap lainnya, seperti DarkRISC-V, PicoRV32, dan VexRISC-V, jauh lebih cepat daripada *core* yang diteliti. *Soft processor core* RISC-V tersebut sangat dioptimalkan dan telah digunakan dalam *FPGA* untuk proyek yang sebenarnya. Selain itu, penulis menganggap peningkatan *throughput* dibandingkan dengan RISC-V *single-cycle* merupakan perkembangan yang minim, karena peningkatan teoretis seharusnya jauh lebih tinggi.

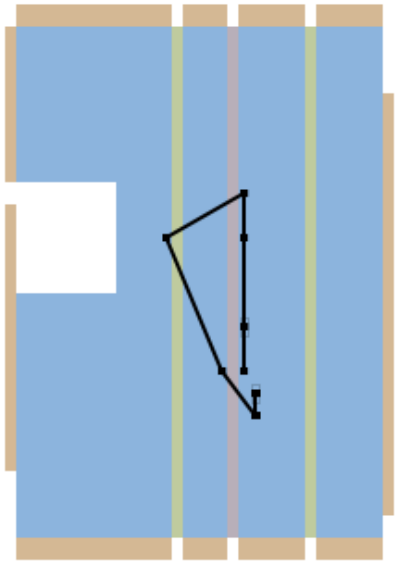
4.4 *Critical Path*

Tabel 4.4: *Path Summary* pada *Timing Report*

From Node	id_ex_regs:IDEX_regs—address_out[3]
To Node	register_files:regfiles0—register_content_rtl_0_bypass[8]
Launch Clock	clk
Latch Clock	clk
Data Arrival Time	22.068
Data Required Time	5.007
Slack	-17.061

Critical path merupakan jalur antara *input* dan *output* dengan *delay* maksimal setelah waktu rangkaian telah dihitung. Jalur kritis dapat dengan mudah ditemukan dengan menggunakan metode traceback. Tabel 4.4 merupakan ringkasan dari analisa *timing* pada *critical datapath*. Jalur terpanjang yang ditemukan yakni jalur dari register *ID/EX* hingga ke register dalam *register files*. Sesuai Tabel 4.5, jalur terpanjang dalam *soft processor* yang dirancang meliputi: register ID/EX, *adder*, *ALU shift right*, pemilihan `ALU_branch_sig`, pemilihan `flush_sig`, hingga *register files*.

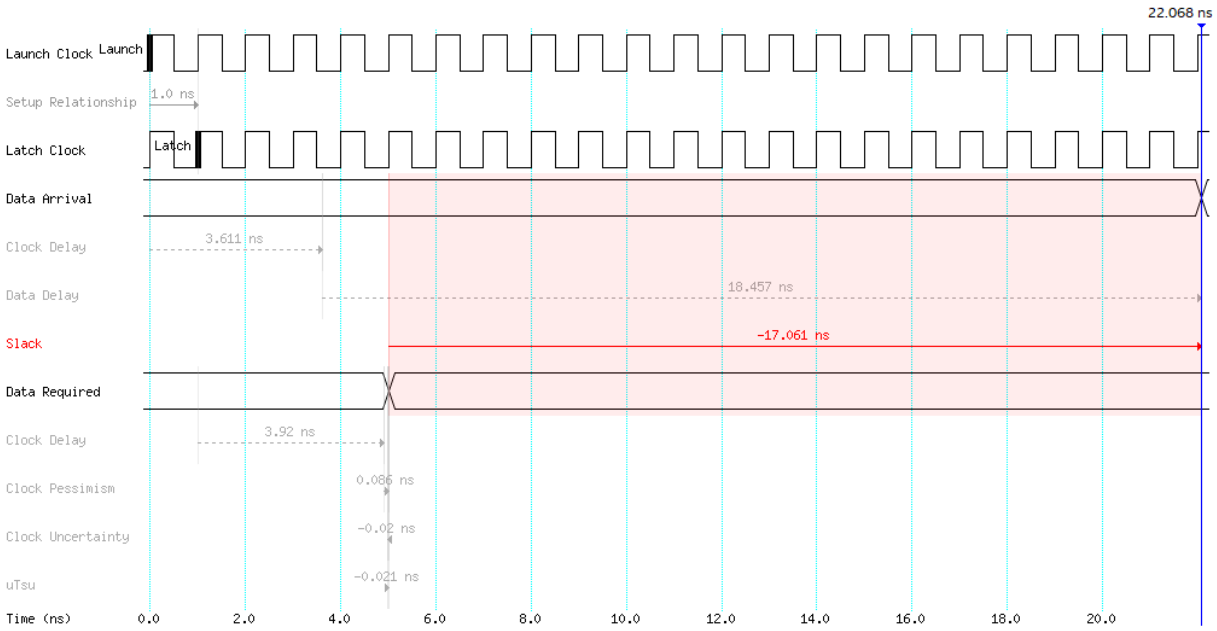
Diagram *timing* dan *delay* pada *critical datapath* dapat dilihat dalam Gambar 4.2. Periode yang merupakan gambaran dari frekuensi maksimal ini dibutuhkan oleh prosesor untuk memastikan proses transmisi *data* dalam prosesor sudah tepat. Semakin panjang rangkaian kombinasional dalam suatu jalur kritis, maka waktu yang diperlukan data untuk tiba di destinasi juga akan meningkat. Pada prosesor yang dirancang, periode ini merupakan periode terpanjang dari kelima tahapan *pipeline*.



Gambar 4.1: Illustrasi grafis dari *datapath* prosesor.

Jalur kritis dari desain disintesis ke dalam bentuk *hardware* seperti pada Gambar 4.1 dalam *FPGA*. Jalur kritis dalam sebuah *FPGA* dihasilkan secara otomatis oleh *compiler* dalam Quartus Prime *Design Software*. Jalur ini dipengaruhi oleh racangan arsitektur prosesor. Semakin panjang lapisan logika kombinasional dalam suatu prosesor, maka

jalur kritis akan menjadi lebih panjang dan mengakibatkan turunnya frekuensi maksimal prosesor. Panjang dari jalur ini dapat dioptimalkan dengan cara membagi logika kombinasional yang panjang menjadi ke beberapa tahapan *pipeline*.



Gambar 4.2: *Timing Waveform* pada *critical path* prosesor.

Tabel 4.5: *Data Arrival Path* pada *Timing Report.*

Total	**Incr**	**RF**	**Type**	**Fanout**	**Location**	**Element**	
0.000	0.000					launch edge time	1
3.611	3.611					clock path	2
3.611	3.611	R				clock network delay	1
22.068	18.457					data path	3
3.872	0.261		uTco	1	FF_X21_Y8_N5	id_ex_regs:IDEX_regs—address_out[3]	1
3.872	0.000	FF	CELL	4	FF_X21_Y8_N5	IDEX_regs—address_out[3]—q	2
4.850	0.978	FF	IC	2	LCCOMB_X21_Y12_N4	Add6∼2—dataa	3
5.402	0.552	FR	CELL	1	LCCOMB_X21_Y12_N4	Add6∼2—cout	4
5.402	0.000	RR	IC	2	LCCOMB_X21_Y12_N6	Add6∼4—cin	5
5.475	0.073	RF	CELL	1	LCCOMB_X21_Y12_N6	Add6∼4—cout	6
5.475	0.000	FF	IC	2	LCCOMB_X21_Y12_N8	Add6∼6—cin	7
5.548	0.073	FR	CELL	1	LCCOMB_X21_Y12_N8	Add6∼6—cout	8
5.548	0.000	RR	IC	2	LCCOMB_X21_Y12_N10	Add6∼8—cin	9
5.621	0.073	RF	CELL	1	LCCOMB_X21_Y12_N10	Add6∼8—cout	10
5.621	0.000	FF	IC	2	LCCOMB_X21_Y12_N12	Add6∼10—cin	11
5.694	0.073	FR	CELL	1	LCCOMB_X21_Y12_N12	Add6∼10—cout	12
5.694	0.000	RR	IC	2	LCCOMB_X21_Y12_N14	Add6∼12—cin	13
5.767	0.073	RF	CELL	1	LCCOMB_X21_Y12_N14	Add6∼12—cout	14
5.767	0.000	FF	IC	2	LCCOMB_X21_Y12_N16	Add6∼14—cin	15
5.840	0.073	FR	CELL	1	LCCOMB_X21_Y12_N16	Add6∼14—cout	16
5.840	0.000	RR	IC	2	LCCOMB_X21_Y12_N18	Add6∼16—cin	17
5.913	0.073	RF	CELL	1	LCCOMB_X21_Y12_N18	Add6∼16—cout	18
5.913	0.000	FF	IC	2	LCCOMB_X21_Y12_N20	Add6∼18—cin	19
5.986	0.073	FR	CELL	1	LCCOMB_X21_Y12_N20	Add6∼18—cout	20
5.986	0.000	RR	IC	2	LCCOMB_X21_Y12_N22	Add6∼20—cin	21
6.059	0.073	RF	CELL	1	LCCOMB_X21_Y12_N22	Add6∼20—cout	22
6.059	0.000	FF	IC	2	LCCOMB_X21_Y12_N24	Add6∼22—cin	23
6.132	0.073	FR	CELL	1	LCCOMB_X21_Y12_N24	Add6∼22—cout	24
6.132	0.000	RR	IC	2	LCCOMB_X21_Y12_N26	Add6∼24—cin	25
6.739	0.607	RR	CELL	1	LCCOMB_X21_Y12_N26	Add6∼24—combout	26
7.883	1.144	RR	IC	1	LCCOMB_X18_Y9_N10	Mux81∼1—datad	27
8.060	0.177	RR	CELL	2	LCCOMB_X18_Y9_N10	Mux81∼1—combout	28
8.323	0.263	RR	IC	1	LCCOMB_X18_Y9_N0	Mux81∼2—datad	29
8.500	0.177	RR	CELL	13	LCCOMB_X18_Y9_N0	Mux81∼2—combout	30
9.486	0.986	RR	IC	1	LCCOMB_X17_Y10_N28	alu0—ShiftRight0∼17—datac	31
9.813	0.327	RR	CELL	1	LCCOMB_X17_Y10_N28	alu0—ShiftRight0∼17—combout	32
11.348	1.535	RR	IC	1	LCCOMB_X23_Y7_N0	alu0—ShiftRight0∼19—datad	33
11.525	0.177	RR	CELL	2	LCCOMB_X23_Y7_N0	alu0—ShiftRight0∼19—combout	34
13.024	1.499	RR	IC	1	LCCOMB_X18_Y12_N18	alu0—ShiftRight0∼22—datad	35
13.201	0.177	RR	CELL	2	LCCOMB_X18_Y12_N18	alu0—ShiftRight0∼22—combout	36
14.786	1.585	RR	IC	1	LCCOMB_X21_Y5_N20	alu0—Mux22∼5—datad	37
14.963	0.177	RR	CELL	1	LCCOMB_X21_Y5_N20	alu0—Mux22∼5—combout	38
15.200	0.237	RR	IC	1	LCCOMB_X21_Y5_N10	alu0—Mux22∼6—datad	39
15.355	0.155	RF	CELL	1	LCCOMB_X21_Y5_N10	alu0—Mux22∼6—combout	40
15.610	0.255	FF	IC	1	LCCOMB_X21_Y5_N12	alu0—Mux22∼9—datad	41
15.749	0.139	FF	CELL	2	LCCOMB_X21_Y5_N12	alu0—Mux22∼9—combout	42
16.004	0.255	FF	IC	1	LCCOMB_X21_Y5_N14	ALU_branch_sig∼1—datad	43
16.172	0.168	FR	CELL	1	LCCOMB_X21_Y5_N14	ALU_branch_sig∼1—combout	44
17.033	0.861	RR	IC	1	LCCOMB_X22_Y8_N30	ALU_branch_sig∼2—datad	45
17.210	0.177	RR	CELL	1	LCCOMB_X22_Y8_N30	ALU_branch_sig∼2—combout	46
17.444	0.234	RR	IC	1	LCCOMB_X22_Y8_N24	ALU_branch_sig∼5—datac	47
17.768	0.324	RR	CELL	1	LCCOMB_X22_Y8_N24	ALU_branch_sig∼5—combout	48
18.005	0.237	RR	IC	1	LCCOMB_X22_Y8_N14	ALU_branch_sig∼6—datad	49
18.182	0.177	RR	CELL	2	LCCOMB_X22_Y8_N14	ALU_branch_sig∼6—combout	50
18.661	0.479	RR	IC	1	LCCOMB_X23_Y8_N10	ALU_branch_sig∼9—datad	51
18.838	0.177	RR	CELL	1	LCCOMB_X23_Y8_N10	ALU_branch_sig∼9—combout	52
19.074	0.236	RR	IC	1	LCCOMB_X23_Y8_N0	ALU_branch_sig∼11—datad	53
19.251	0.177	RR	CELL	2	LCCOMB_X23_Y8_N0	ALU_branch_sig∼11—combout	54
19.516	0.265	RR	IC	1	LCCOMB_X23_Y8_N22	flush_sig—datad	55
19.693	0.177	RR	CELL	56	LCCOMB_X23_Y8_N22	flush_sig—combout	56
20.869	1.176	RR	IC	1	LCCOMB_X26_Y9_N12	IFID_regs—instruction_out∼7—datad	57
21.024	0.155	RF	CELL	3	LCCOMB_X26_Y9_N12	IFID_regs—instruction_out∼7—combout	58
21.814	0.790	FF	IC	1	LCCOMB_X26_Y8_N8	regfiles0—register_content_rtl_0_bypass[8]∼feeder—datad	59
21.953	0.139	FF	CELL	1	LCCOMB_X26_Y8_N8	regfiles0—register_content_rtl_0_bypass[8]∼feeder—combout	60
21.953	0.000	FF	IC	1	FF_X26_Y8_N9	regfiles0—register_content_rtl_0_bypass[8]—d	61
22.068	0.115	FF	CELL	1	FF_X26_Y8_N9	register_files:regfiles0—register_content_rtl_0_bypass[8]	62

4.5 Pengujian Program Deret Fibonacci

Penulis menggunakan sejumlah algoritma deret Fibonacci dalam pengujian, yakni algoritma Fibonacci iteratif dan algoritma Fibonacci rekursif. Meski keduanya berfungsi untuk melakukan kalkulasi deret Fibonacci, mereka memiliki ciri khas masing-masing. Algoritma Fibonacci iteratif jauh lebih sederhana dari varian yang rekursif. Prosesor membutuhkan jauh lebih banyak memori dan tenaga komputasi dalam penghitungan deret Fibonacci secara rekursif.

4.5.1 Algoritma Deret Fibonacci Iteratif yang Digunakan

Listing 4.1: Program perulangan deret Fibonacci dari yang pertama hingga bilangan Fibonacci terakhir di bawah 10.000.

```
    lui x15 0x2
    addi x10 x15 1807
start:
    addi x6,x0,1
    addi x5,x0, 0
up:
    add x7,x5,x6
    bge x7,x10,start
    sw x7,0x34(x0)
    sw x6,0x30(x0)
    lw x5,0x30(x0)
    sw x7,0x30(x0)
    lw x6,0x30(x0)
    jal up
```

Tabel 4.6: Nilai yang tersimpan dalam *Program Memory*

Address	***Stored Value***
0	00000000
4	000027b7
8	70f78513
12	00100313
16	00000293
20	006283b3
24	fea3dae3
28	02702c23
32	02602a23
36	03402283
40	02702a23
44	03402303
48	fe5ff0ef

Listing 4.2: Penulisan program dalam bentuk heksadesimal ke *Program Memory*.

```
type rom_type is array (0 to 255) of std_logic_vector (31 downto 0);
signal ROM : rom_type:=(
  -- Example program
  x"00000000",
```

```
    x"000027b7",
    x"70f78513",
    x"00100313",
    x"00000293",
    x"006283b3",
    x"fea3dae3",
    x"02702c23",
    x"02602a23",
    x"03402283",
    x"02702a23",
    x"03402303",
    x"fe5ff0ef",
    others => x"00000000"
);
```

Program yang digunakan untuk pengujian yakni program deret Fibonacci. Algoritma program ini melakukan penghitungan nilai deret Fibonacci satu per satu dari 1 hingga bilangan Fibonacci dibawah 10.000. Hasil dari kalkulasi disimpan dalam sebuah register dalam *Register Files*. Pertama-tama, algoritma dituliskan ke dalam bentuk RISC-V Assembly. Kemudian, kode Assembly tersebut de-*assemble* menggunakan RISC-V GCC *toolchain*. Hasil yang didapatkan yakni instruksi yang sudah dalam bentuk biner atau heksadesimal. Listing 4.1 merupakan algoritma dalam RISC-V Assembly yang digunakan. Sedangkan, Tabel 4.6 merupakan hasil de-*assembly* dari program perulangan deret Fibonacci. Program dituliskan ke dalam bentuk *Hardware Description Language* pada *file* VHDL *Program Memory* seperti pada Listing 4.2.

4.5.2 Algoritma Deret Fibonacci Rekursif yang Digunakan

Listing 4.3: Program kalkulasi bilangan Fibonacci secara rekursif dalam bahasa C.

```
asm("auipc sp 0x1");  // Set stack pointer
asm("jal main");          // call main
asm("NOP");         // End of Program

int fibbonacci(int n) {
   if(n == 0){
      return 0;
   } else if(n == 1) {
      return 1;
   } else {
      return (fibbonacci(n-1) + fibbonacci(n-2));
   }
}

int main() {
    int x = 0;
     x = fibbonacci(5);
    return 0;
}
```

Listing 4.4: Program penghitungan deret Fibonacci secara rekursif.

```
auipc x2 0x1
jal x1 <main>
addi x11 x10 0
addi x17 x0 10
```

```
5  addi x0 x0 0
6
7  <fibbonacci>:
8      addi x2 x2 -32
9      sw x1 28 x2
10     sw x8 24 x2
11     sw x9 20 x2
12     addi x8 x2 32
13     sw x10 -20 x8
14     lw x15 -20 x8
15     bne x15 x0 12
16     addi x15 x0 0
17     jal x0 68
18     lw x14 -20 x8
19     addi x15 x0 1
20     bne x14 x15 12
21     addi x15 x0 1
22     jal x0 48
23     lw x15 -20 x8
24     addi x15 x15 -1
25     addi x10 x15 0
26     jal x1 <fibbonacci>
27     addi x9 x10 0
28     lw x15 -20 x8
29     addi x15 x15 -2
30     addi x10 x15 0
31     jal x1 <fibbonacci>
32     addi x15 x10 0
33     add x15 x9 x15
34     addi x10 x15 0
35     lw x1 28 x2
36     lw x8 24 x2
37     lw x9 20 x2
38     addi x2 x2 32
39     jalr x0 x1 0
40
41 <main>:
42     addi x2 x2 -32
43     sw x1 28 x2
44     sw x8 24 x2
45     addi x8 x2 32
46     sw x0 -20 x8
47     addi x10 x0 10
48     auipc x1 0x0
49     jalr x1 x1 -152
50     sw x10 -20 x8
51
52 <stop>:
53     nop
54     jal <stop>
```

Soft Processor yang dirancang juga mengujikan penghitungan nilai bilangan Fibonacci ke-`N` menggunakan metode rekursif. Tidak seperti varian iteratif, program rekursif ini memerlukan prosesor untuk memanfaatkan *data memory* dan *stack pointer*. Pertamatama, program dituliskan dalam bahasa C seperti pada Listing 4.3. Kemudian, program dikompilasi ke dalam bahasa Assembly RISC-V menggunakan GCC *toolchain*. Hasil dari

Tabel 4.7: Nilai yang tersimpan dalam *program memory*.

Address	***Stored Value***
0	00000000
4	40000113
8	090000ef
12	00050593
16	00a00893
20	00000013
24	fe010113
28	00112e23
32	00812c23
36	00912a23
40	02010413
44	fea42623
48	fec42783
52	00079663
56	00000793
60	0440006f
64	fec42703
68	00100793
72	00f71663
76	00100793
80	0300006f
84	fec42783
88	fff78793
92	00078513
96	fb9ff0ef
100	00050493
104	fec42783
108	ffe78793
112	00078513
116	fa5ff0ef
120	00050793
124	00f487b3
128	00078513
132	01c12083
136	01812403
140	01412483
144	02010113
148	00008067
152	fe010113
156	00112e23
160	00812c23
164	02010413
168	fe042623
172	00c00513
176	00000097
180	f68080e7
184	fea42623
188	00000013

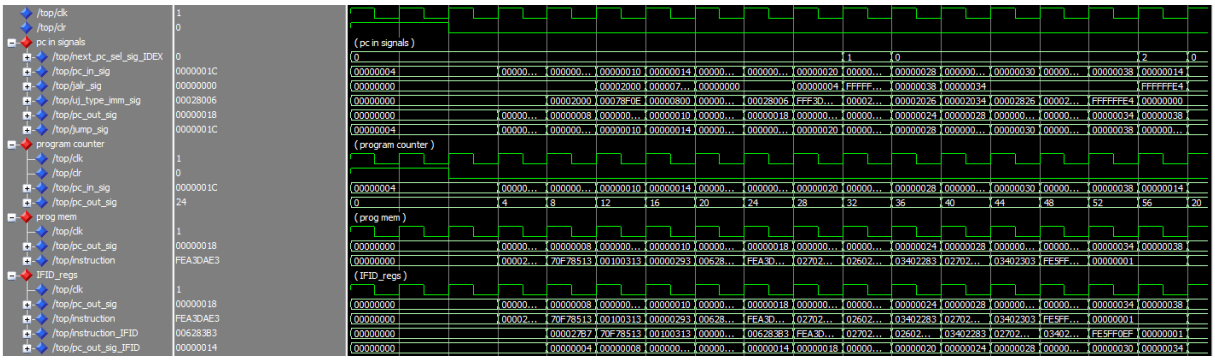
kompilasi tersebut kemudian dituliskan ke dalam *program memory* seperti pada Tabel 4.7 untuk kemudian dijalankan oleh *soft processor*.

Listing 4.4 merupakan algoritma deret Fibonacci rekursif dalam bahasa Assembly RISC-V. Sedangkan, prosesor menjalankan program dengan bentuk bahasa mesin RISC-V dalam *program memory* seperti pada Tabel 4.7. *Compiler* mula-mula mengkompilasi program yang dituliskan dalam bahasa Assembly ke dalam bentuk bahasa mesin RISC-V. Kemudian, simulator ModelSim menjalankan program yang disimpan di dalam *Program Memory soft processor*. Pertama-tama, program melakukan pengaturan alamat tertinggi *stack pointer*. Pengguna dapat menyesuaikan alamat yang diperlukan sesuai dengan program dan kapabilitas *FPGA*. Kemudian, instruksi `JAL` melakukan *jump* ke bagian program yang berlabel `<main>`, sama seperti program dalam bahasa C. Kemudian, bagian `<main>` menjalankan *subroutine* penghitungan bilangan Fibonacci ke-n yang berlabel `<fibonacci>`. Prosesor menyimpan nilai balikkan dari fungsi rekursif menggunakan *data memory* dengan memanfaatkan register `x2` yang merupakan register *return address*.

4.6 Simulasi Program Deret Fibonacci Iteratif

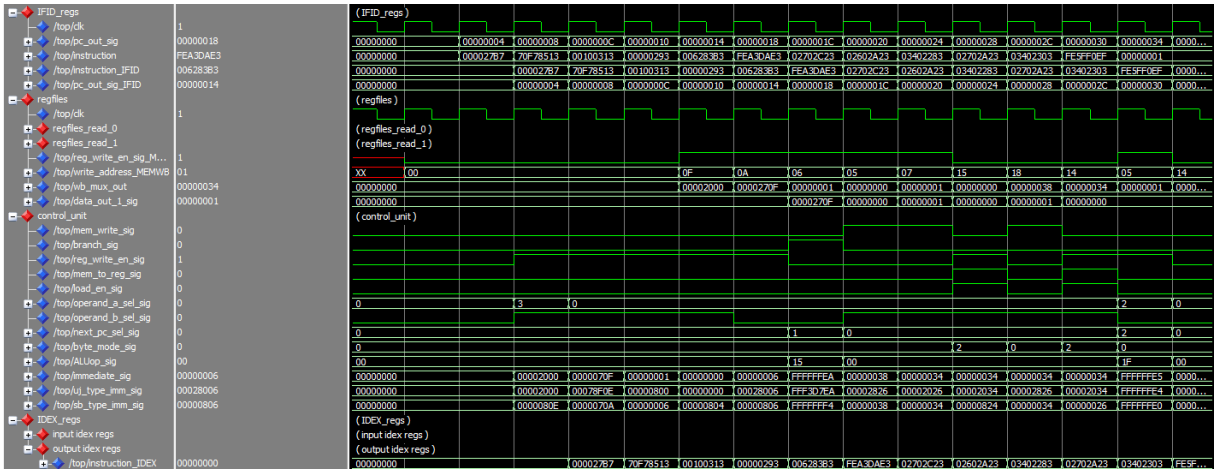
Prosesor memproses setiap instruksi dalam lima tahap *pipeline* yang berbeda. Tahap-tahap ini, diantaranya: *instruction fetch*, *instruction decode*, *execution*, *memory*, dan *write back*. Setiap komponen dalam arsitektur *soft processor* yang diajukan mengolah masukan menjadi keluaran sesuai dengan tipe instruksi dan sinyal kendali. Hasil dari satu tahap akan dilanjutkan ke tahap berikutnya pada *clock* selanjutnya. Setiap instruksi membutuhkan sekitar 5 siklus *clock*. Sejumlah instruksi membutuhkan siklus yang lebih, karena adanya proses *stall* dan *flush* dalam pencegahan *hazard*.

4.6.1 *Instruction Fetch Stage*



Gambar 4.3: *Waveform* sinyal pada tahap *pipeline Instruction Fetch*.

Tahap *Instruction Fetch* ditunjukkan dalam Gambar 4.3. Terdapat sejumlah sinyal yang berlangsung dalam tahap ini. Diantaranya sinyal yang berhubungan dengan *Program Counter*, *Program Memory*, dan *input* dari register *IF/ID*. Nilai yang masuk ke dalam Program Counter adalah hasil keluaran dari `pc_in_sig`, yang dipilih oleh sinyal kontrol `next_pc_sel_sig_IDEX` yang berasal dari tahap *Execution*. Keluaran *program memory* menjadi *input* register *pipeline IF/ID*.

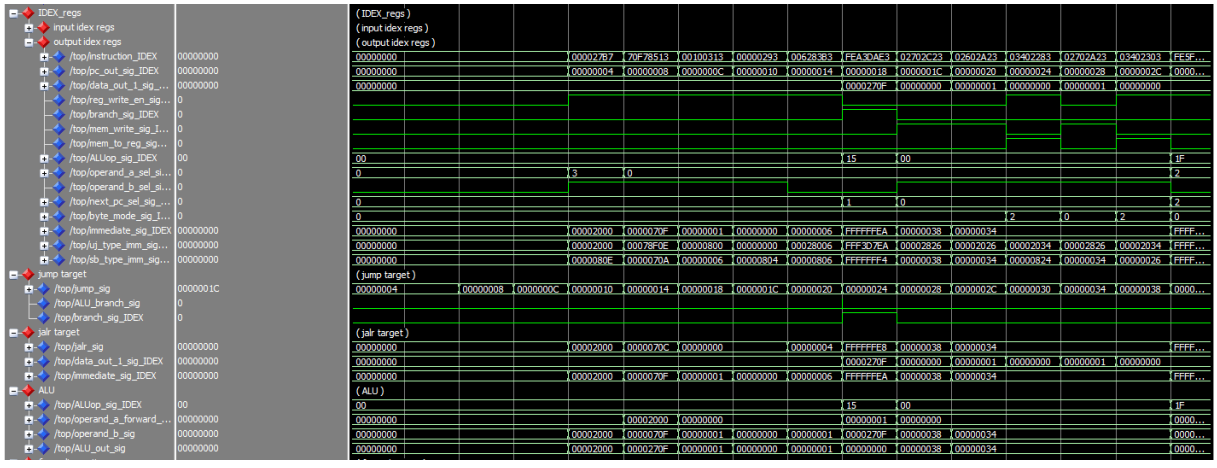


Gambar 4.4: *Waveform* sinyal pada tahap *pipeline Instruction Decode*.

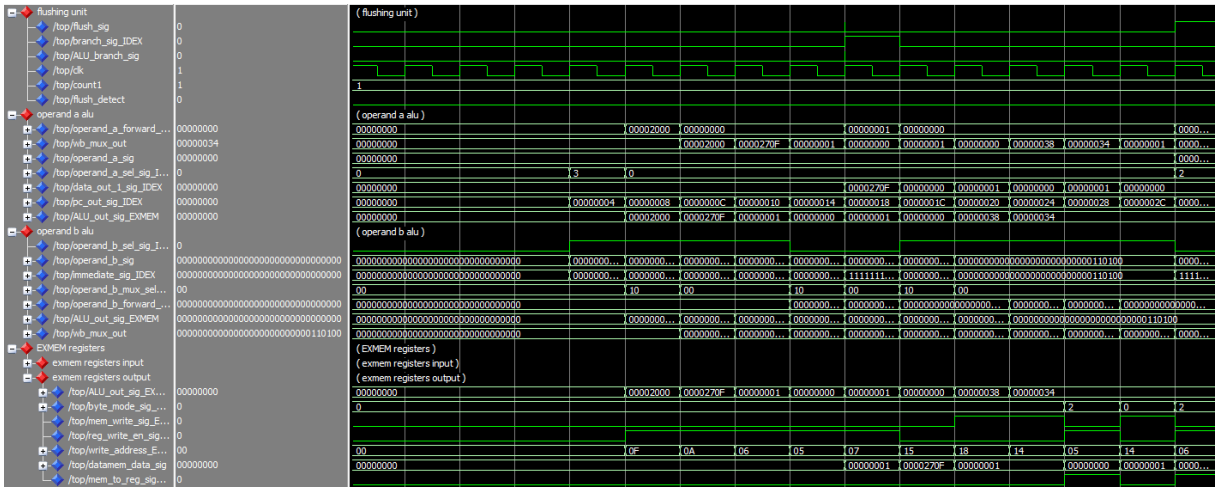
4.6.2 *Instruction Decode Stage*

Waveform sinyal pada tahap *Instruction Decode* ditunjukkan dalam Gambar 3.12. Tahap ini menginterpretasi instruksi dari tahap *Instruction Fetch* dan menghasilkan sinyal kontrol yang sesuai. Selain itu, dilakukan pembacaan dan penulisan terhadap register dalam *Register Files*. Sinyal yang menjadi masukan *register files* berasal dari multiplekser pada tahap *Write Back*. Sinyal kontrol yang dihasilkan *control unit*, sejumlah sinyal dari tahap sebelumnya, serta hasil pembacaan *register files* diteruskan ke dalam tahap *Execution* melalui register *pipeline ID/EX*.

4.6.3 *Execution Stage*



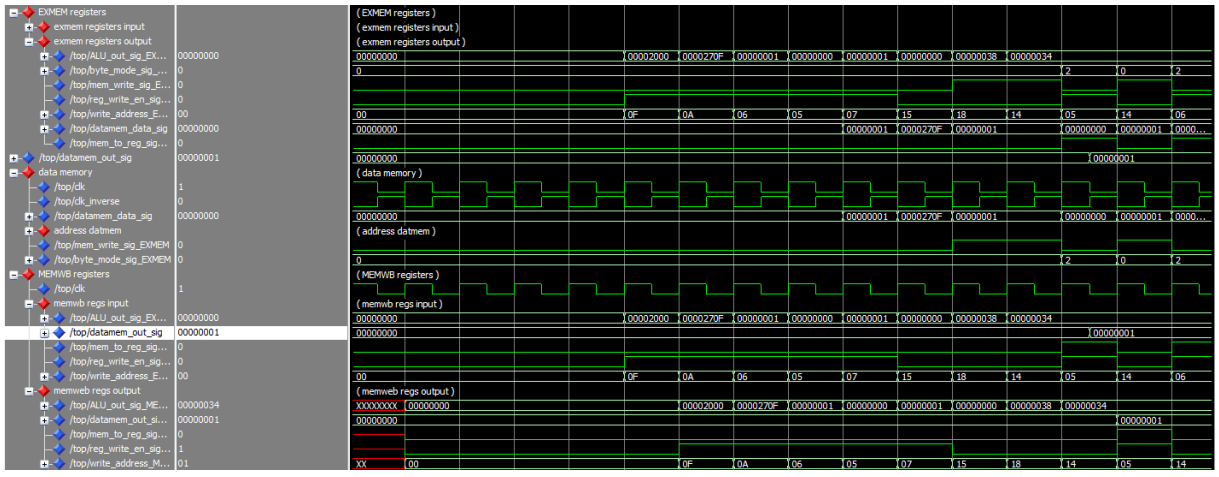
Gambar 4.5: *Waveform* sinyal pada tahap *pipeline Execution*.



Gambar 4.6: *Waveform* sinyal pada tahap *pipeline Execution*.

Gambar 4.5 dan Gambar 4.6 merupakan kumpulan *waveform* dari sinyal yang berlangsung dalam tahap *Execution*. Dalam tahap ini, sinyal kontrol dari tahap *Instruction Decode* yang diteruskan oleh register *pipeline ID/EX* mengendalikan keluaran dan masukan *Arithmetic Logic Unit*. *Operand* dari *ALU* dipilih oleh serangkaian multiplekser yang dikendalikan oleh *forwarding unit* dan sinyal kontrol dari tahap sebelumnya. Selain itu, dalam tahap ini juga dilakukan pemeriksaan terhadap peristiwa *jump*, *branch taken*, dan *hazard*. Sinyal hasil kalkulasi dalam *ALU* serta sejumlah sinyal kontrol diteruskan ke tahap *Memory* menggunakan register *pipeline EX/MEM*.

4.6.4 *Memory Stage*

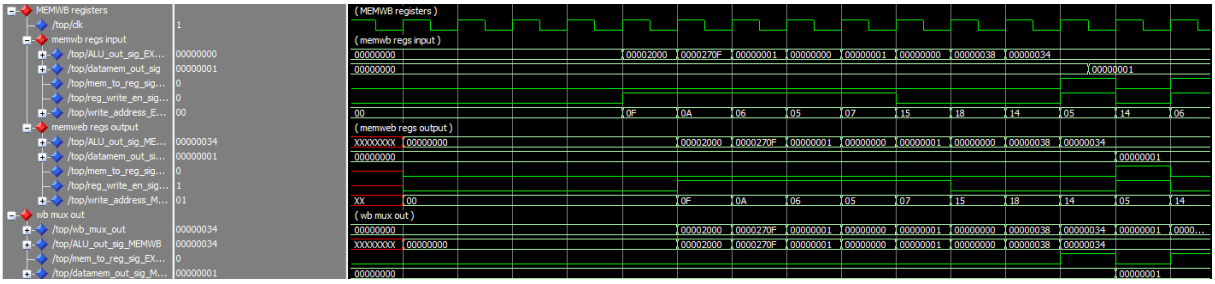


Gambar 4.7: *Waveform* sinyal pada tahap *pipeline Memory*.

Dalam tahap ini, dilakukan penulisan nilai *input* ke *data memory* pada saat sinyal kontrol bernilai aktif. Alamat memori, data masukan, serta sinyal kontrol yang menjadi masukan *data memory* diteruskan dari tahap *Execution* melalui register *pipeline EX/MEM*. Karena RISC-V merupakan arsitektur *Load/Store*, maka hanya kedua instruksi tersebut saja lah yang mengubah nilai yang tersimpan dalam *Data Memory*. Nilai hasil kalkulasi *ALU* dan nilai bacaan dari *data memory* diteruskan ke tahap *Write Back*

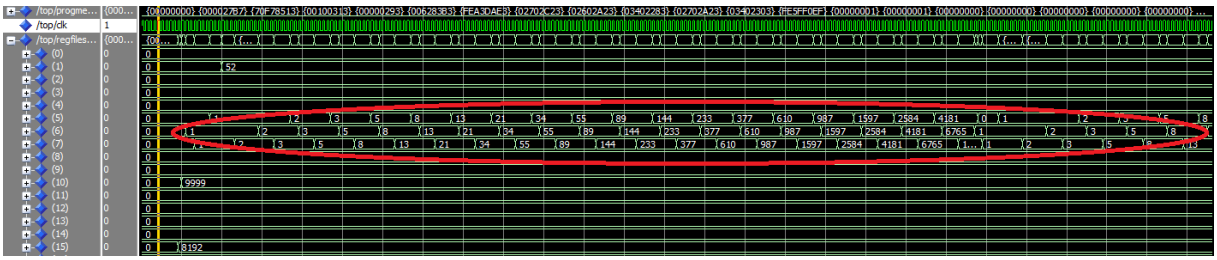
melalui register *pipeline MEM/WB*. Sinyal-sinyal yang terdapat dalam tahap *Memory* ditunjukkan dalam Gambar 4.7.

4.6.5 *Write Back Stage*



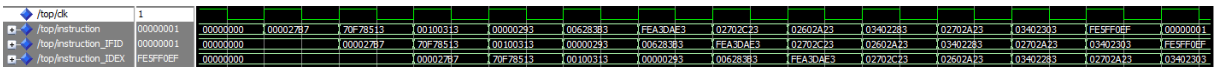
Gambar 4.8: *Waveform* sinyal pada tahap *pipeline Write Back*.

4.6.6 Hasil Deret Fibonacci Iteratif



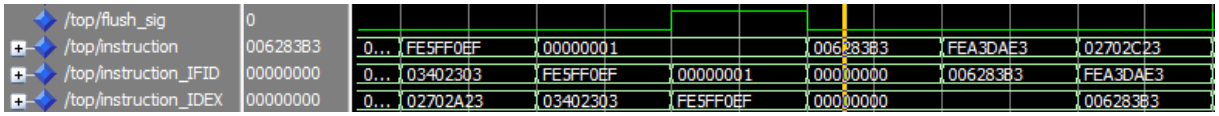
Gambar 4.9: Perubahan nilai yang disimpan dalam *Register Files* selama program berlangsung.

Hasil yang tersimpan dalam *Register Files*, seperti pada Gambar 4.9, menunjukkan bahwa program sudah dieksekusi oleh *soft processor* dengan berhasil. Ini ditandakan dari perubahan nilai pada register `x5`, `x6`, dan `x7` setiap beberapa *clock cycle*. Pada saat `x7` bernilai lebih dari 9999, maka pada *clock* berikutnya operasi BGE (*Branch Greater Equal*) menghasilkan *branch taken*. Alhasil, program melakukan *branching* ke alamat *program memory* yang berlabel `start`.

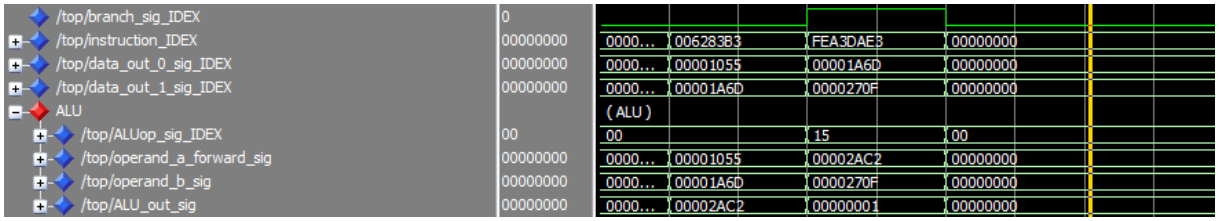


Gambar 4.10: Instruksi dari *program memory* dilanjutkan ke tahap *pipeline* selanjutnya pada setiap *clock*.

Instruksi dari *program memory* diteruskan ke tahapan *pipeline* berikutnya dalam menjalankan program. Pada gambar 4.10, instruksi satu per satu diteruskan dari tahap *Instruction Fetch*, *Instruction Decode*, hingga *Execution*. Keseluruhan nilai pada pipeline tahap IF/ID dan tahap ID/EX dihapus pada saat suatu *branch* atau *jump* terjadi. Peristiwa yang dinamakan sebagai *flushing* ini bertujuan supaya data terbaru dalam *pipeline* tersebut tidak keliru akibat *hazard*. Gambar 4.11 menunjukkan peristiwa *flushing* pada *potive edge clock* berikutnya ketika `flush_sig = 1`. Pada program yang dijalankan, instruksi yang dapat mengakibatkan *flushing* yakni `jal up` dan `bge x7 , x10 , start` pada saat *branch taken*.



Gambar 4.11: Pada saat *branch* atau *jump* terdeteksi, prosesor melakukan *flushing*. Nilai dalam register *pipeline* dihapus pada *positive edge clock* berikutnya.

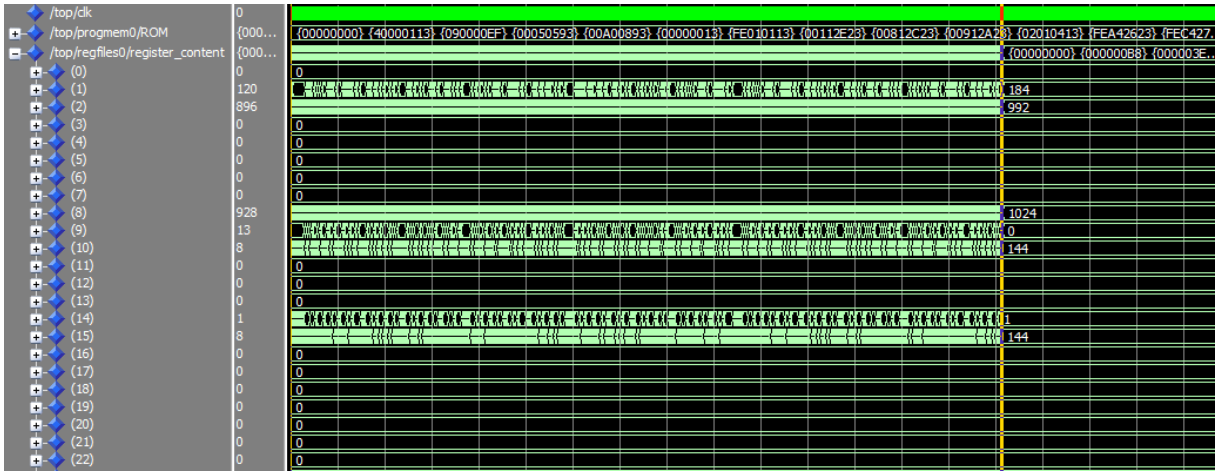


Gambar 4.12: *Branch* dideteksi dengan menggunakan *Arithmetic Logic Unit*.

ALU digunakan untuk melakukan deteksi kapan *branch* perlu dilakukan. Dalam kasus ini, nilai *operand* A dibandingkan dengan nilai *operand* B. Gambar 4.12 merupakan kondisi *ALU* pada stage *Execution* saat `x7` lebih besar dari 9999. Instruksi `bge x7 , x10 , start` yang berjalan dalam *stage* tersebut menghasilkan keluaran bernilai 1 pada *output ALU*. Nilai ini kemudian digunakan untuk pemilihan keputusan oleh multiplekser yang memilih nilai *Program Counter* berikutnya.

4.6.7 Hasil Deret Fibonacci Rekursif

GCC *compiler* menggunakan register `x2` *stack pointer* secara *default*. Alamat memori 1000_{16} digunakan sebagai alamat tertinggi *stack* pada awal mula program. *Stack Pointer* perlu diatur sedemikian rupa sesuai dengan kebutuhan program untuk menghindari *overflow*. Setelah itu, program melakukan *jump* ke bagian *main* dari algoritma. Prosesor secara berulang kali menjalankan subrutin Fibonacci rekursif dimulai dari bilangan `N` yang dideklarasikan hingga mencapai *base case*.

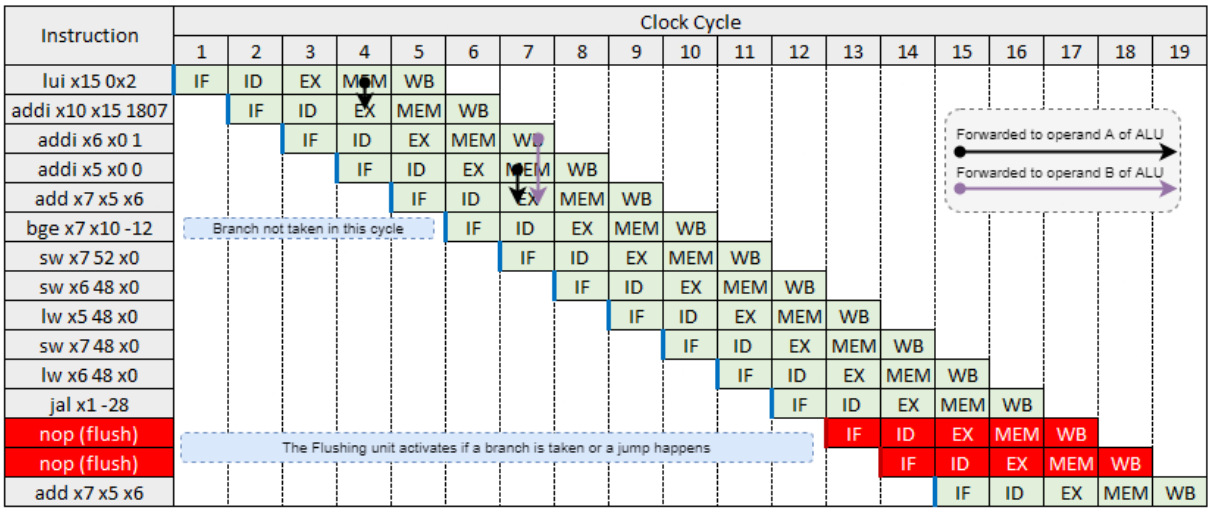


Gambar 4.13: Hasil eksekusi program kalkulasi bilangan Fibonacci secara rekursif.

Prosesor menyelesaikan komputasi nilai ke-12 bilangan Fibonacci secara rekursif setelah sekitar 15.000 *clock cycles*. Tidak seperti algoritma yang iteratif, yang berjalan dengan

kompleksitas linear, varian rekursif ini memerlukan jauh lebih banyak tenaga komputasi dan sumber daya karena sifat kompleksitas kuadratiknya. Hasil ini dapat dilihat dalam Gambar 4.13.

4.7 Pengujian *Forwarding Unit*



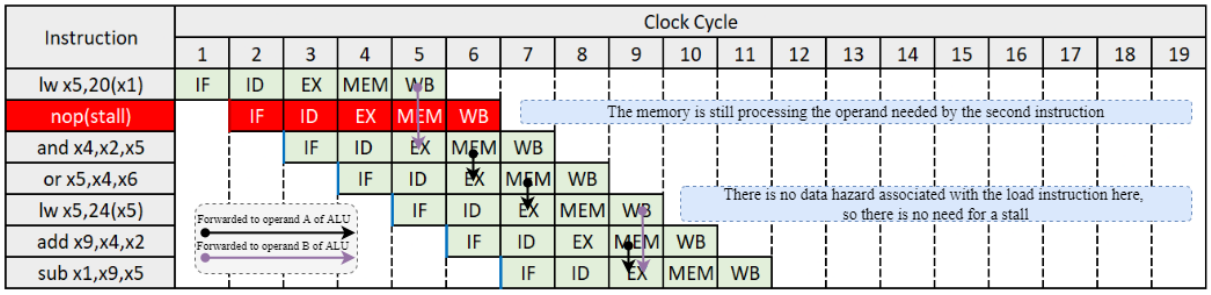
Gambar 4.14: Eksekusi dari program pengujian deret Fibonacci iteratif dalam Listing 4.1.

Iterasi pertama dari program deret Fibonaci iteratif diilustrasikan pada Gambar 4.14. Penulis menggunakan beberapa panah untuk menandakan *forwarding* terhadap *operand*. Misalnya, pada instruksi pertama dan kedua, *operand* pertama dari instruksi `addi x10 x15 1807` belum ditulis ulang kembali ke *register files*. Untuk menjaga hasil yang benar, prosesor harus melakukan *forwarding* data dari tahap *Memory* dari instruksi sebelumnya. Hal serupa terjadi antara instruksi kelima (`add x7 x5 x6`) dengan instruksi ketiga (`addi x6 x0 1`) dan instruksi keempat (`addi x5 x0 0`). Di antara instruksi ketiga dan kelima, prosesor perlu meneruskan hasil perhitungan register `x6` pada tahap *Write Back* ke *operand* kedua *ALU*. Antara instruksi keempat dan kelima, prosesor perlu meneruskan hasil *ALU* di tahap *Memory* ke *operand* pertama *ALU*. Operasi keenam dari program (`bge x7 x10 -12`) akan memulai ulang program dan semua register yang disimpan. Pada perulangan pertama, program akan berjalan seperti biasa, karena `x7` masih kurang dari `x10`. Ketika *jump* atau *branching* terdeteksi, seperti operasi `jal x1 -28`, unit *flushing* akan menghapus konten register *pipeline IF/ID* dan *ID/EX*. Ini akan menyebabkan dua siklus berikutnya dieksekusi sebagai `NOP`.

4.8 Pengujian *Hazard Detection Unit*

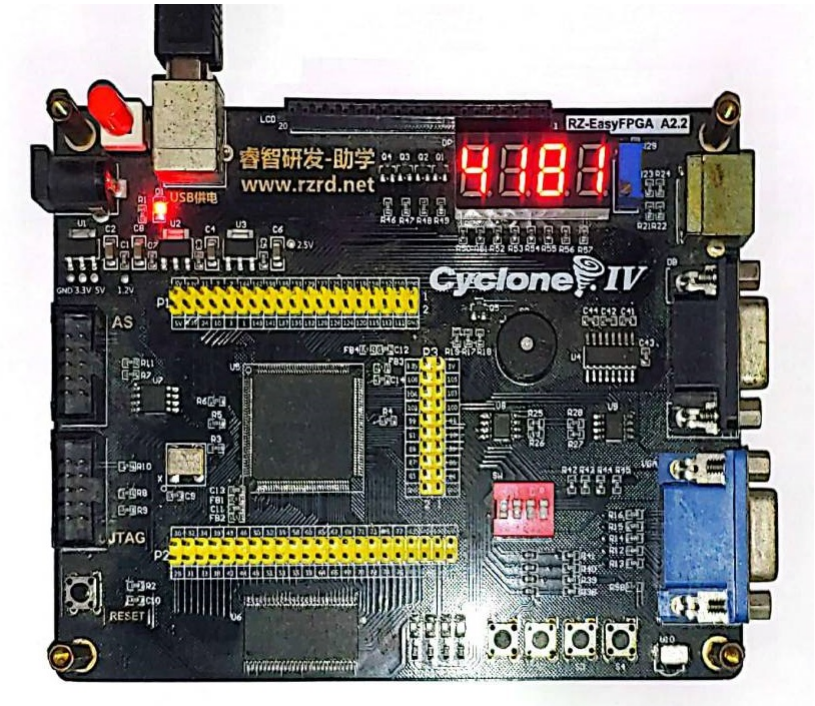
Listing 4.5: Program pengujian *Hazard Detection Unit*.

```
1 lw x5 ,20( x1)
2 and x4 ,x2 ,x5
3 or x5 ,x4 ,x6
4 lw x5 ,24( x2)
5 add x9 ,x4 ,x2
6 sub x1 ,x6 ,x7
```



Gambar 4.15: Eksekusi dari program pengujian *Hazard Detection Unit* pada Listing 4.5.

4.9 Pengujian pada *FPGA* Aktual



Gambar 4.16: *Processor core* menjalankan program dalam *Program Memory* setelah diunggah ke *FPGA*.

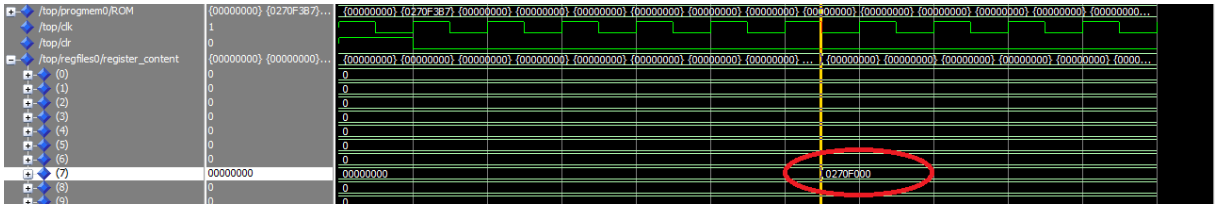
FPGA telah berhasil menjalankan *soft processor* hasil rancangan beserta program pengujian. Pertama-tama, penulis mengunggah *soft processor* beserta dengan program dengan menggunakan Quartus Prime. Setelah proses sintesis dan pengunggahan selesai, FPGA dapat menjalankan program dalam *program memory*. Gambar 4.16 menampilkan angka 4181, yang merupakan bilangan Fibonacci. Selain itu, *FPGA* juga telah menampilkan bilangan Fibonacci lain di bawah 9999. *Soft Processor* dalam FPGA mengeksekusi instruksi dalam *program memory* satu per satu dan menyimpan hasilnya dalam *register files*. Kemudian, *seven segment* menampilkan bilangan tersebut.

4.10 Pengujian Instruksi-Instruksi RV32I

1. LUI

Listing 4.6: Program pengujian instruksi LUI.

```
0:     0270f3b7     lui  x7  0x270f
```



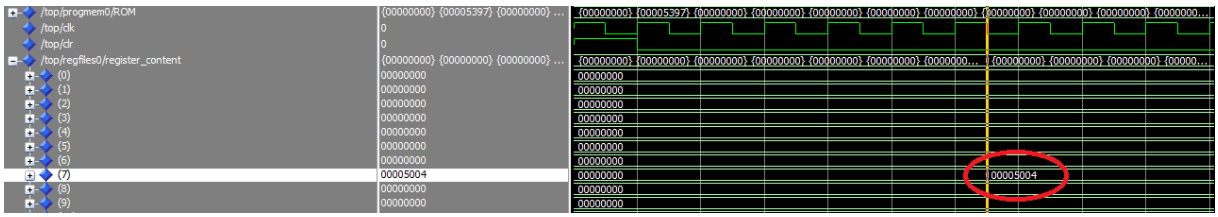
Gambar 4.17: Hasil pengujian instruksi LUI.

Listing 4.6 merupakan program yang digunakan untuk menguji instruksi `LUI`. Gambar 4.17 merupakan hasil pengujian *soft processor* dalam menjalankan program pengujian instruksi `LUI` menggunakan *software* simulasi ModelSim. Pada program pengujian, instruksi `lui x7 0x270f` mengisi 20 bit teratas register `x7` dengan nilai verb—270f—$_{16}$. Hasil ini terlihat dalam perubahan nilai register `x7` beberapa *clock cycle* setelah program dijalankan.

2. AUIPC

Listing 4.7: Program pengujian instruksi AUIPC.

```
0:     00005397     auipc  x7  0x5
```



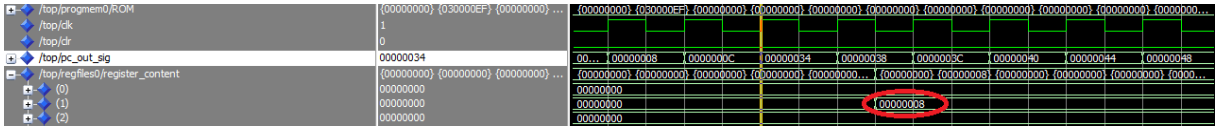
Gambar 4.18: Hasil pengujian instruksi AUIPC.

Listing 4.7 merupakan program yang digunakan untuk menguji instruksi `AUIPC`. Gambar 4.18 merupakan hasil pengujian *soft processor* dalam menjalankan program pengujian instruksi `AUIPC` menggunakan *software* simulasi ModelSim. Instruksi —auipc x7 0x5— yang diujikan digunakan untuk membangun nilai register `x7` dengan menjumlahkan *Program Counter* dengan suatu nilai *immediate*. *Soft processor* telah berhasil menjumlahkan *immediate* tipe-U dengan nilai yang relatif dengan *Program Counter*. Hasil ini dapat dilihat pada register `x7` setelah tahap *Writeback*.

3. JAL

Listing 4.8: Program pengujian instruksi JAL.

```
0:     030000ef     jal  x1  48
```



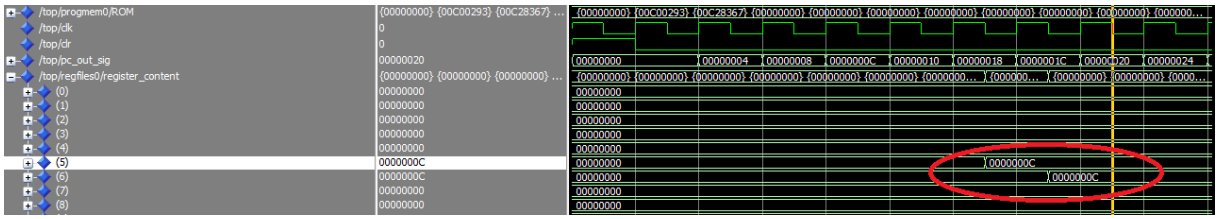
Gambar 4.19: Hasil pengujian instruksi JAL.

Listing 4.8 merupakan program yang digunakan untuk menguji instruksi `JAL`. Gambar 4.19 merupakan hasil pengujian *soft processor* dalam menjalankan program pengujian instruksi `JAL` menggunakan *software* simulasi ModelSim. Operasi `jal x1 48` digunakan untuk mengubah alamat *Program Counter* menjadi `48` dengan cara *jump*, kemudian meletakkan *return address* berupa nilai *Program Counter* saat ini ditambah dengan 4. *Return Address* pada instruksi ini diletakkan pada register `x1`. Dalam hasil yang didapatkan, *soft processor* mampu menjalankan instruksi `JAL` sesuai dengan spesifikasi.

4. JALR

Listing 4.9: Program pengujian instruksi JALR.

```
0:      00c00293      addi  x5  x0  12
4:      00c28367      jalr  x6  x5  12
```



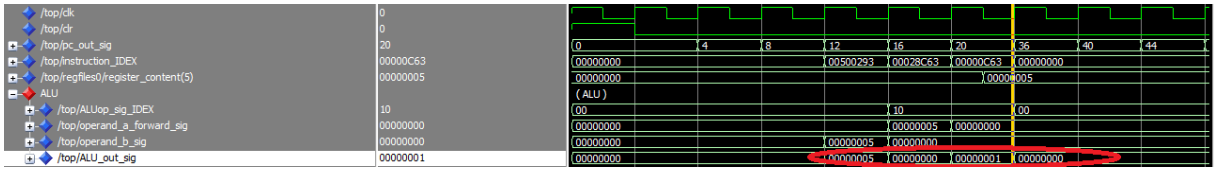
Gambar 4.20: Hasil pengujian instruksi JALR.

Listing 4.9 merupakan program yang digunakan untuk menguji instruksi `JALR`. Gambar 4.20 merupakan hasil pengujian *soft processor* dalam menjalankan program pengujian instruksi `JALR` menggunakan *software* simulasi ModelSim. Berbeda dengan instruksi `JAL` yang hanya menggunakan *immediate*, instruksi `JALR` menggunakan nilai *offset* dari *immediate* dan juga *register*. Pada program pengujian, pertama-tama `addi x5 x0 12` mengubah nilai dalam register `x5` menjadi `12`. Pada *clock cycle* selanjutnya, `jalr x6 x5 12` menjumlahkan nilai baru `x5` dengan nilai *immediate* `12`. Karena pada *clock cycle* tersebut nilai belum tersimpan dalam *Register Files*, *operand* berupa nilai register `x5` di-*forward* dari tahap *pipeline* tempat nilai `x5` terbaru berada. Hasil dalam gambar menunjukkan bahwa prosesor sudah tepat dalam menjalankan serangkaian instruksi ini.

5. BEQ

Listing 4.10: Program pengujian instruksi BEQ.

```
0:      00500293      addi  x5  x0  5
4:      00028c63      beq   x5  x0  24
8:      00000c63      beq   x0  x0  24
```



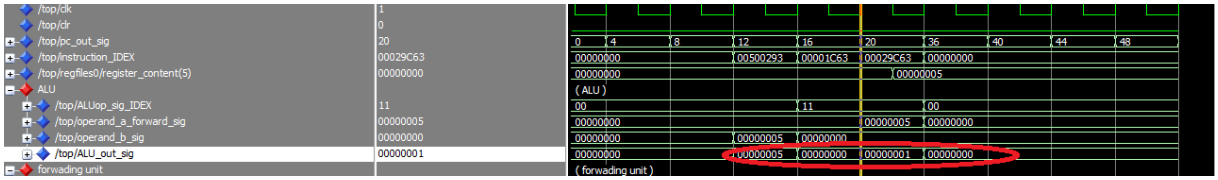
Gambar 4.21: Hasil pengujian instruksi BEQ.

Listing 4.10 merupakan program yang digunakan untuk menguji instruksi `BEQ`. Gambar 4.21 merupakan hasil pengujian *soft processor* dalam menjalankan program pengujian instruksi `BEQ` menggunakan *software* simulasi ModelSim. Operasi *branching* `BEQ` melakukan lompatan ke alamat target jika nilai *operand* pertama sama dengan nilai *operand* kedua. Dalam program pengujian yang dibuat, telah diuji dua instruksi `BEQ` yang berbeda. Pada instruksi `BEQ` yang pertama, *branch* tidak terjadi karena nilai `x5` yang sudah bernilai `5` tidak sama dengan nilai `x0` yang selalu bernilai `0`. Sedangkan, seperti pada gambar hasil, operasi kedua menghasilkan *branch taken* ke alamat `24` karena nilai *operand* pertama `x0` sama dengan nilai *operand* kedua `x0`.

6. BNE

Listing 4.11: Program pengujian instruksi BNE.

```
0:      00500293      addi x5 x0 5
4:      00001c63      bne x0 x0 24
8:      00029c63      bne x5 x0 24
```



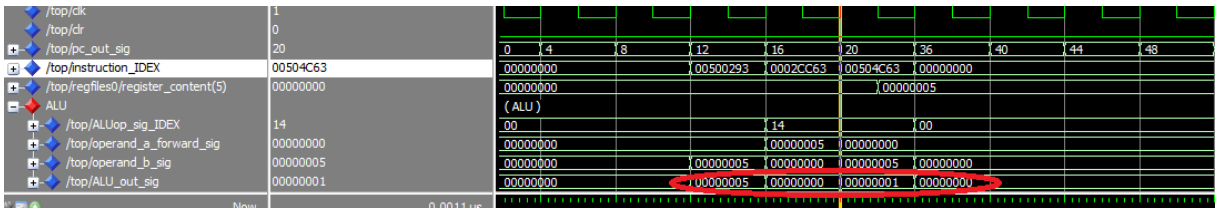
Gambar 4.22: Hasil pengujian instruksi BNE.

Listing 4.11 merupakan program yang digunakan untuk menguji instruksi `BNE`. Gambar 4.22 merupakan hasil pengujian *soft processor* dalam menjalankan program pengujian instruksi `BNE` menggunakan *software* simulasi ModelSim. Operasi `BNE` merupakan kebalikan dari operasi `BEQ` di mana *branching* dilakukan pada saat nilai *operand* pertama tidak sama dengan *operand* kedua. Operasi `BNE` pertama tidak menghasilkan *branch taken* karena nilai *operand* pertama dan kedua sama. Sedangkan, pada operasi `bne x5 x0 24` terjadi peristiwa *branch taken*.

7. BLT

Listing 4.12: Program pengujian instruksi BLT.

```
0:      00500293      addi x5 x0 5
4:      0002cc63      blt x5 x0 24
8:      00504c63      blt x0 x5 24
```



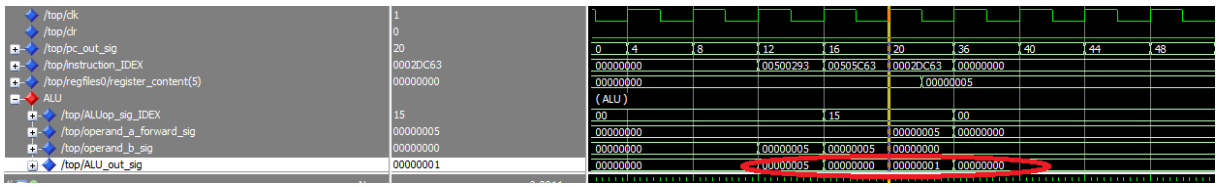
Gambar 4.23: Hasil pengujian instruksi BLT.

Listing 4.12 merupakan program yang digunakan untuk menguji instruksi `BLT`. Gambar 4.23 merupakan hasil pengujian *soft processor* dalam menjalankan program pengujian instruksi `BLT` menggunakan *software* simulasi ModelSim. Untuk melakukan *branching* pada saat nilai *operand* pertama *ALU* lebih kecil daripada nilai *operand* kedua, digunakan perintah `BLT`. Pada saat operasi `blt x5 x0 24` dijalankan, nilai register `x5` yang di-*forward* yakni `5`. Karena nilai ini lebih besar daripada `0`, maka *branch* tidak diambil. Sedangkan, pada instruksi `blt x0 x5 24`, terjadi *branching* ke alamat `24`.

8. BGE

Listing 4.13: Program pengujian instruksi BGE.

```
0:      00500293      addi  x5  x0  5
4:      00505c63      bge   x0  x5  24
8:      0002dc63      bge   x5  x0  24
```



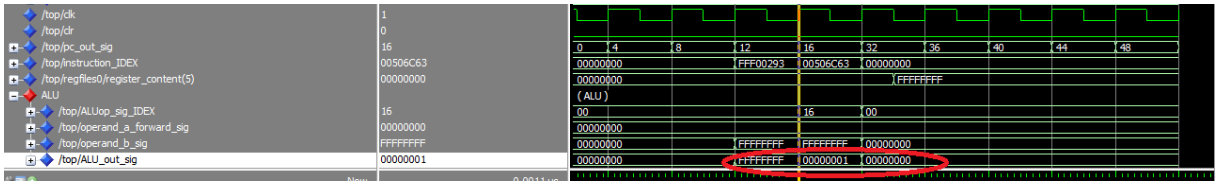
Gambar 4.24: Hasil pengujian instruksi BGE.

Listing 4.13 merupakan program yang digunakan untuk menguji instruksi `BGE`. Gambar 4.24 merupakan hasil pengujian *soft processor* dalam menjalankan program pengujian instruksi `BGE` menggunakan *software* simulasi ModelSim. `BGE` merupakan operasi *branching* pada RISC-V *ISA* yang membuat *program counter* melompat ke alamat tertentu jika nilai *operand* pertama *ALU* lebih besar atau sama dengan *operand* kedua *ALU*. Dalam skenario pengujian, didapatkan bahwa *soft processor* telah mampu melakukan *branching* pada ke alamat *program counter* `24` pada saat operand pertama register `x5` lebih besar dari operand kedua `x0`. Pada kasus yang sebaliknya, *branching* tidak terjadi.

9. BLTU

Listing 4.14: Program pengujian instruksi BLTU.

```
0:      fff00293      addi  x5  x0  -1
4:      00506c63      bltu  x0  x5  24
8:      0002ec63      bltu  x5  x0  24
```



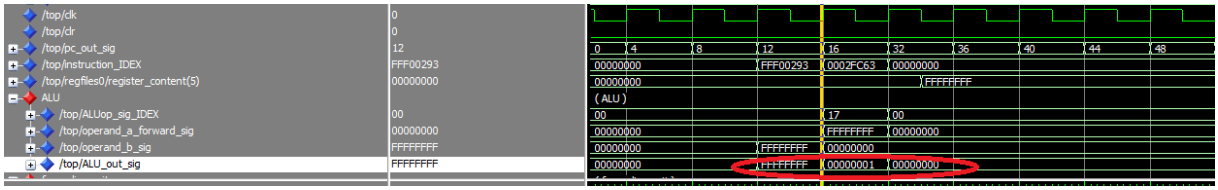
Gambar 4.25: Hasil pengujian instruksi BLTU.

Listing 4.14 merupakan program yang digunakan untuk menguji instruksi `BLTU`. Gambar 4.25 merupakan hasil pengujian *soft processor* dalam menjalankan program pengujian instruksi `BLTU` menggunakan *software* simulasi ModelSim. Sama seperti operasi `BLT`, `BLTU` melakukan percabangan ke alamat *program counter* tertentu pada saat *operand* pertama *ALU* lebih kecil nilainya daripada *operand* kedua *ALU*. Bedanya, pada instruksi `BLTU` konvensi pembacaan nilai *operand* yakni bersifat *unsigned*. Artinya, *Most Significant Bit* pada bilangan *operand* tidak diberlakukan sebagai *sign-bit*, melainkan sebagai suatu bilangan biasa. Pada konvensi pembacaan *unsigned*, operasi `addi x5 x0 -1` menghasilkan bilangan terbesar 32-bit, yakni $\texttt{FFFFFFFF}_{16}$. Ini mengakibatkan *branch taken* pada operasi `bltu x0 x5 24` di mana nilai `0` pada *operand* pertama lebih kecil daripada nilai $\texttt{FFFFFFFF}_{16}$ pada *operand kedua*.

10. BGEU

Listing 4.15: Program pengujian instruksi BGEU.

```
0:      fff00293      addi  x5  x0  -1
4:      0002fc63      bgeu  x5  x0  24
8:      00507c63      bgeu  x0  x5  24
```



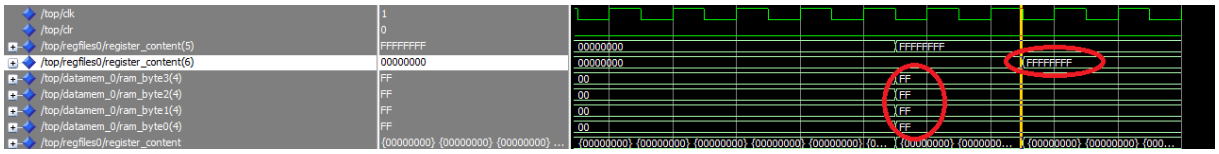
Gambar 4.26: Hasil pengujian instruksi BGEU.

Listing 4.15 merupakan program yang digunakan untuk menguji instruksi `BGEU`. Gambar 4.26 merupakan hasil pengujian *soft processor* dalam menjalankan program pengujian instruksi `BGEU` menggunakan *software* simulasi ModelSim. Operasi `BGEU` merupakan varian *unsigned* dari operasi `BGE`. Pada program pengujian, register `x5` diatur nilainya menjadi $\texttt{FFFFFFFF}_{16}$. Pada konvensi pembacaan *signed*, nilai ini merupakan nilai paling negatif dari bilangan 32-bit. Sedangkan pada konvensi pembacaan *unsigned*, nilai ini merupakan nilai yang terbesar. Oleh karena itu, seperti pada hasil pengujian, operasi `bgeu x5 x0 24` telah menyebabkan *branching* ke alamat `24`.

11. LB

Listing 4.16: Program pengujian instruksi LB.

```
0:      fff00293      addi  x5  x0  -1
4:      00502823      sw    x5  16  x0
8:      01000303      lb    x6  16  x0
```



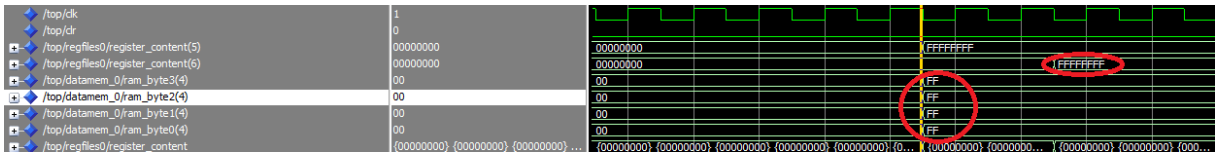
Gambar 4.27: Hasil pengujian instruksi LB.

Listing 4.16 merupakan program yang digunakan untuk menguji instruksi `LB`. Gambar 4.27 merupakan hasil pengujian *soft processor* dalam menjalankan program pengujian instruksi `LB` menggunakan *software* simulasi ModelSim. Untuk memuat nilai yang tersimpan dalam suatu alamat *data memory* ke register dalam *register files*, digunakan operasi *Load*. Salah satu instruksi *Load* dalam RV32I *ISA* yakni `LB`. Operasi ini memuat nilai *byte* yang di-*sign-extend* ke bilangan 32-bit yang tersimpan dalam suatu alamat memori. Nilai ini kemudian dituliskan ke register target pada tahap *Writeback*. Ini sudah sesuai dengan hasil pengujian, di mana nilai *byte* dengan alamat awal `16` yang dibaca di-*sign-extend* ke dalam bentuk 32-bit-nya sebelum dituliskan ke register `x6`.

12. LH

Listing 4.17: Program pengujian instruksi LH.

```
0:      fff00293      addi  x5  x0  -1
4:      00502823      sw    x5  16  x0
8:      01001303      lh    x6  16  x0
```



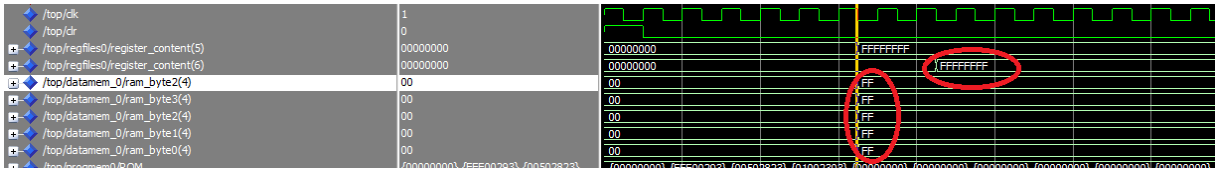
Gambar 4.28: Hasil pengujian instruksi LH.

Listing 4.17 merupakan program yang digunakan untuk menguji instruksi `LH`. Gambar 4.28 merupakan hasil pengujian *soft processor* dalam menjalankan program pengujian instruksi `LH` menggunakan *software* simulasi ModelSim. `LH` memuat nilai *Halfword* dari suatu alamat *data memory* ke register tujuan dalam *register files*. Pembacaan nilai *halfword* dibaca secara *signed*, di mana nilai pembacaan melalui process *sign-extension*. Hasil pengujian yang didapatkan sudah sesuai dengan spesifikasi yang dibutuhkan.

13. LW

Listing 4.18: Program pengujian instruksi LW.

```
0:      fff00293      addi  x5  x0  -1
4:      00502823      sw    x5  16  x0
8:      01002303      lw    x6  16  x0
```



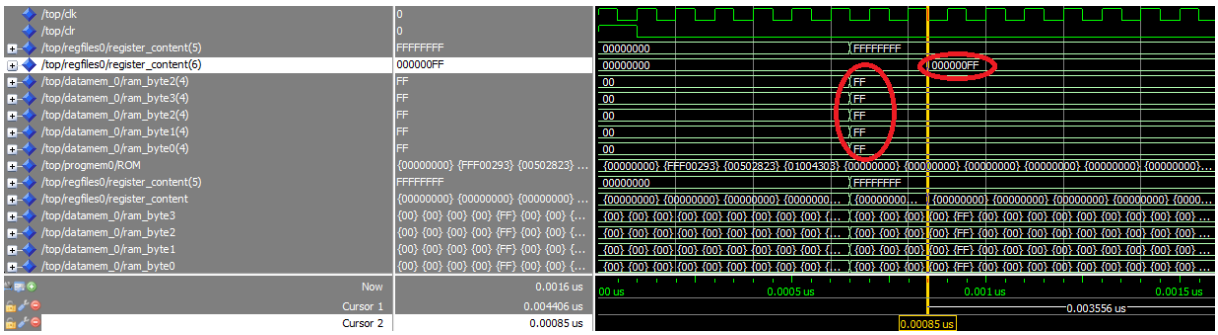
Gambar 4.29: Hasil pengujian instruksi LW.

Listing 4.18 merupakan program yang digunakan untuk menguji instruksi `LW`. Gambar 4.29 merupakan hasil pengujian *soft processor* dalam menjalankan program pengujian instruksi `LW` menggunakan *software* simulasi ModelSim. Operasi *Load Word* memuat nilai *word* yakni 4 *byte* atau 2 *halfword* dari alamat pembacaan *data memory* ke alamat tujuan register. Nilai yang dibaca tidak perlu di-*sign-extend* lagi karena sudah berukuran 32-bit. Pada hasil pengujian, didapatkan bahwa *soft processor* telah dapat memuat nilai *word* yang tersimpan dalam alamat dasar `16` ke register tujuan `x6` pada tahap *Writeback*.

14. LBU

Listing 4.19: Program pengujian instruksi LBU.

```
0:      fff00293      addi x5 x0 -1
4:      00502823      sw x5 16 x0
8:      01004303      lbu x6 16 x0
```



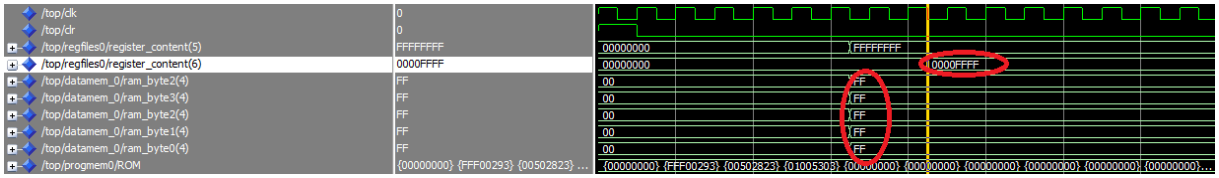
Gambar 4.30: Hasil pengujian instruksi LBU.

Listing 4.19 merupakan program yang digunakan untuk menguji instruksi `LBU`. Gambar 4.30 merupakan hasil pengujian *soft processor* dalam menjalankan program pengujian instruksi `LBU` menggunakan *software* simulasi ModelSim. Instruksi `LBU` merupakan varian *unsigned* dari instruksi *Load Byte*. Nilai *byte* yang dibaca tidak mengalami proses *sign-extension*. Hal ini dapat dilihat dalam hasil pengujian, di mana nilai *byte* hasil pembacaan memori langsung dituliskan ke register tujuan pada tahap *Writeback*.

15. LHU

Listing 4.20: Program pengujian instruksi LHU.

```
0:      fff00293      addi x5 x0 -1
4:      00502823      sw x5 16 x0
8:      01005303      lhu x6 16 x0
```



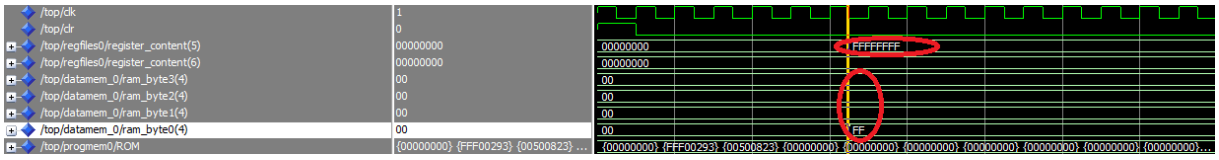
Gambar 4.31: Hasil pengujian instruksi LHU.

Listing 4.20 merupakan program yang digunakan untuk menguji instruksi `LHU`. Gambar 4.31 merupakan hasil pengujian *soft processor* dalam menjalankan program pengujian instruksi `LHU` menggunakan *software* simulasi ModelSim. Sama seperti instruksi `LBU`, operasi *Load Halfword Unsigned* memuat nilai dalam *data memory* sepanjang 16-bit ke register tujuan tanpa melakukan *sign-extension*. Hasil pengujian telah menunjukkan proses pembacaan *halfword unsigned* dari *data memory* dan penulisan ke register tujuan yang tepat.

16. SB

Listing 4.21: Program pengujian instruksi SB.

```
0:      fff00293      addi x5 x0 -1
4:      00500823      sb x5 16 x0
```



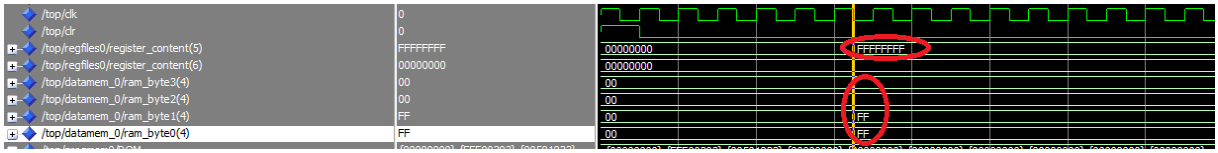
Gambar 4.32: Hasil pengujian instruksi SB.

Listing 4.21 merupakan program yang digunakan untuk menguji instruksi `SB`. Gambar 4.32 merupakan hasil pengujian *soft processor* dalam menjalankan program pengujian instruksi `SB` menggunakan *software* simulasi ModelSim. Bertolak belakang dengan *Load*, *Store* merupakan instruksi dalam RV32I yang memuat nilai dari suatu register dalam *register files* ke dalam suatu alamat tujuan *data memory*. `SB` atau *Store Byte* merupakan instruksi *Store* yang menyimpan nilai *byte*, atau 8-bit terbawah dari register asal ke alamat *data memory* tujuan. Nilai yang dibaca bersifat *unsigned* di mana tidak dilakukan *sign-extension*. Hasil pengujian menunjukkan bahwa 8-bit terrendah dari register `x5` telah dituliskan ke alamat *data memory* `16`.

17. SH

Listing 4.22: Program pengujian instruksi SH.

```
0:      fff00293      addi x5 x0 -1
4:      00501823      sh x5 16 x0
```



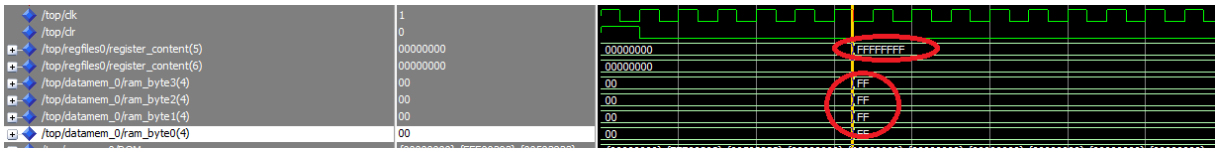
Gambar 4.33: Hasil pengujian instruksi SH.

Listing 4.22 merupakan program yang digunakan untuk menguji instruksi `SH`. Gambar 4.33 merupakan hasil pengujian *soft processor* dalam menjalankan program pengujian instruksi `SH` menggunakan *software* simulasi ModelSim. Instruksi `SH` memuat nilai *16-bit* terendah, atau *halfword*, dari register asal ke alamat *data memory* tujuan. Operasi `sh x5 16 x0` yang diujikan telah memuat nilai *halfword* dari register asal `x5` ke alamat *data memory* `16`.

18. SW

Listing 4.23: Program pengujian instruksi SW.

```
0:      fff00293      addi x5 x0 -1
4:      00502823      sw x5 16 x0
```



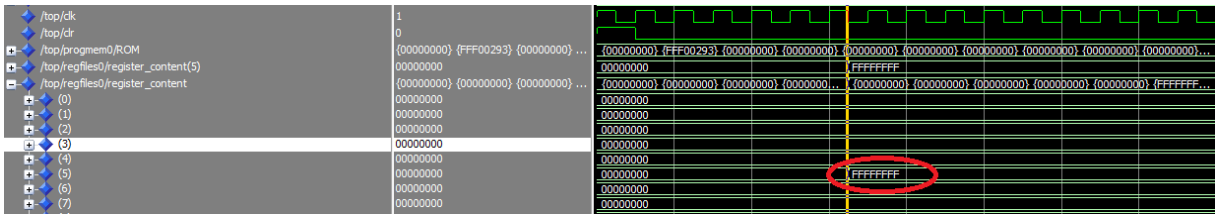
Gambar 4.34: Hasil pengujian instruksi SW.

Listing 4.23 merupakan program yang digunakan untuk menguji instruksi `SW`. Gambar 4.34 merupakan hasil pengujian *soft processor* dalam menjalankan program pengujian instruksi `SW` menggunakan *software* simulasi ModelSim. Operasi *Store* terakhir `SW` menyimpan nilai *word* dari register asal dalam *register files* ke dalam alamat *data memory* tujuan. Instruksi `sw x5 16 x0` yang diujikan telah memuat nilai 32-bit $\texttt{FFFFFFFF}_{16}$ dalam register `x5` ke alamat *data memory* `16`.

19. ADDI

Listing 4.24: Program pengujian instruksi ADDI.

```
0:      fff00293      addi x5 x0 -1
```



Gambar 4.35: Hasil pengujian instruksi ADDI.

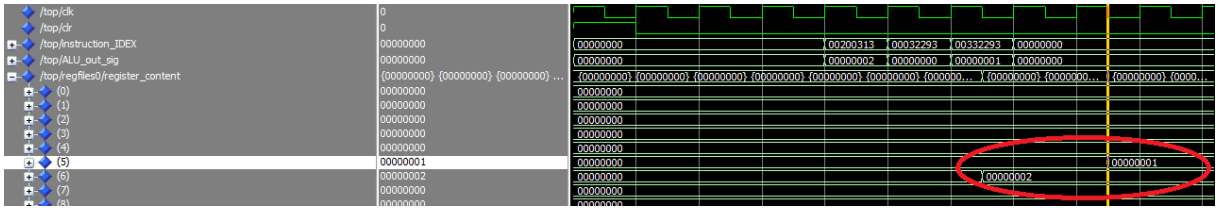
Listing 4.24 merupakan program yang digunakan untuk menguji instruksi `ADDI`. Gambar 4.35 merupakan hasil pengujian *soft processor* dalam menjalankan program

pengujian instruksi `ADDI` menggunakan *software* simulasi ModelSim. Instruksi `ADDI` menjumlahkan nilai *operand* pertama yang berasal dari *register files* dengan suatu nilai *immediate*. Instruksi `addi x5 x0 -1` yang diujikan telah berhasil menjumlahkan nilai dalam register `x5` yang masih bernilai `0` dengan *immediate* `-1`. Hasilnya yakni $\texttt{FFFFFFFF}_{16}$ yang merupakan bilangan terbesar 32-bit.

20. SLTI

Listing 4.25: Program pengujian instruksi SLTI.

```
0:      00200313      addi  x6  x0  2
4:      00032293      slti  x5  x6  0
8:      00332293      slti  x5  x6  3
```



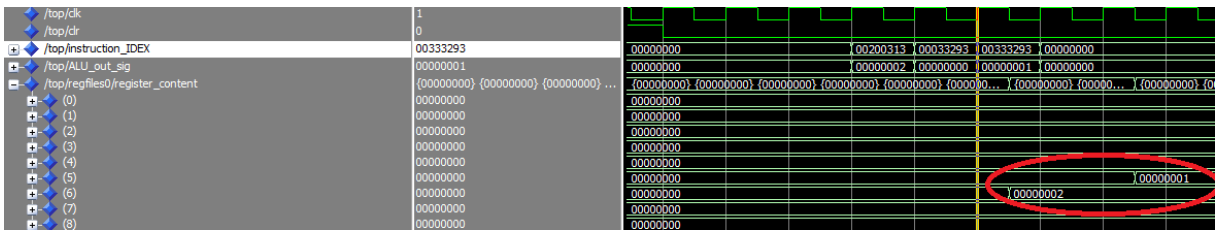
Gambar 4.36: Hasil pengujian instruksi SLTI.

Listing 4.25 merupakan program yang digunakan untuk menguji instruksi `SLTI`. Gambar 4.36 merupakan hasil pengujian *soft processor* dalam menjalankan program pengujian instruksi `SLTI` menggunakan *software* simulasi ModelSim. *Set Less Than Immediate* merupakan instruksi RV32I yang memuat nilai $\texttt{00000001}_{16}$ ke dalam register tujuan pada saat nilai *operand* pertama *ALU* lebih kecil daripada nilai *operand* kedua. Dari hasil yang diujikan, operasi `slti x5 x6 0` telah menyimpan nilai $\texttt{00000001}_{16}$ ke dalam register `x5` karena nilai `2` dala register `x6` lebih kecil daripada `3`.

21. SLTIU

Listing 4.26: Program pengujian instruksi SLTIU.

```
0:      00200313      addi  x6  x0  2
4:      00033293      sltiu  x5  x6  0
8:      00333293      sltiu  x5  x6  3
```



Gambar 4.37: Hasil pengujian instruksi SLTIU.

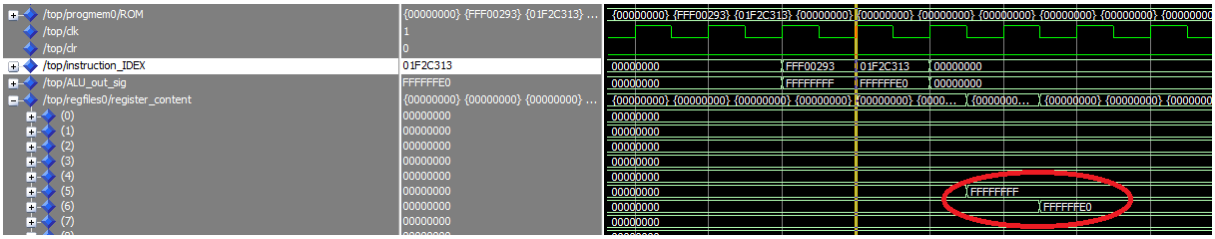
Listing 4.26 merupakan program yang digunakan untuk menguji instruksi `SLTIU`. Gambar 4.37 merupakan hasil pengujian *soft processor* dalam menjalankan program pengujian instruksi `SLTIU` menggunakan *software* simulasi ModelSim. Operasi ini

merupakan varian *unsigned* dari `SLT`. Nilai *operand* dan *immediate* yang menjadi *input ALU* dibaca secara *unsigned*. Karena *operand* yang digunakan tidak mengalami *overflow*, maka hasil pengujian instruksi `SLTIU` yang didapatkan sama seperti hasil pengujian `SLT` sebelumnya.

22. XORI

Listing 4.27: Program pengujian instruksi XORI.

```
0:      fff00293      addi x5 x0 -1
4:      01f2c313      xori x6 x5 31
```



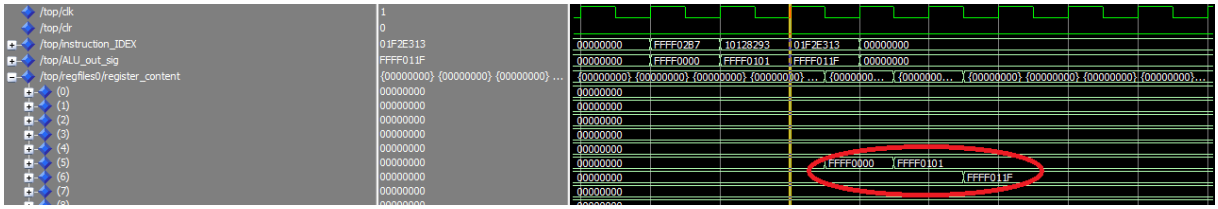
Gambar 4.38: Hasil pengujian instruksi XORI.

Listing 4.27 merupakan program yang digunakan untuk menguji instruksi `XORI`. Gambar 4.38 merupakan hasil pengujian *soft processor* dalam menjalankan program pengujian instruksi `XORI` menggunakan *software* simulasi ModelSim. Instruksi `XORI` melakukan operasi XOR pada nilai register asal terhadap nilai *immediate* yang telah di-*sign-extend*. Pada pengujian, nilai $\texttt{FFFFFFFF}_{16}$ dalam register `x5` telah di XOR-kan dengan nilai `31` dari *immediate*. Hasilnya kemudian disimpan dalam register tujuan `x6` pada tahap *pipeline* Writeback.

23. ORI

Listing 4.28: Program pengujian instruksi ORI.

```
0:      ffff02b7      lui x5 0xffff0
4:      10128293      addi x5 x5 257
8:      01f2e313      ori x6 x5 31
```



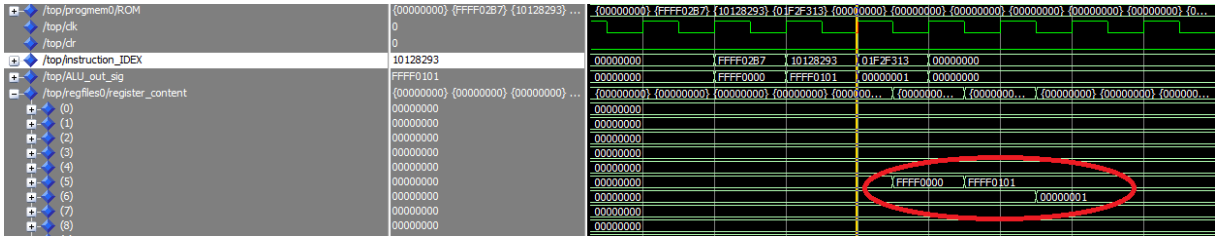
Gambar 4.39: Hasil pengujian instruksi ORI.

Listing 4.28 merupakan program yang digunakan untuk menguji instruksi `ORI`. Gambar 4.39 merupakan hasil pengujian *soft processor* dalam menjalankan program pengujian instruksi `ORI` menggunakan *software* simulasi ModelSim. `ORI` merupakan instruksi yang melakukan operasi logika OR kepada nilai dalam register asal terdadap nilai *immediate*. Dari hasil pengujian, didapatkan bahwa instruksi `ori x6 x5 31` telah mampu melakukan operasi logika OR antara nilai dalam register asal `x5` dengan nilai *immediate* `31` dan menyimpan hasilnya dalam register `x6`.

24. ANDI

Listing 4.29: Program pengujian instruksi ANDI.

```
0:      ffff02b7      lui  x5  0xffff0
4:      10128293      addi  x5  x5  257
8:      01f2f313      andi  x6  x5  31
```



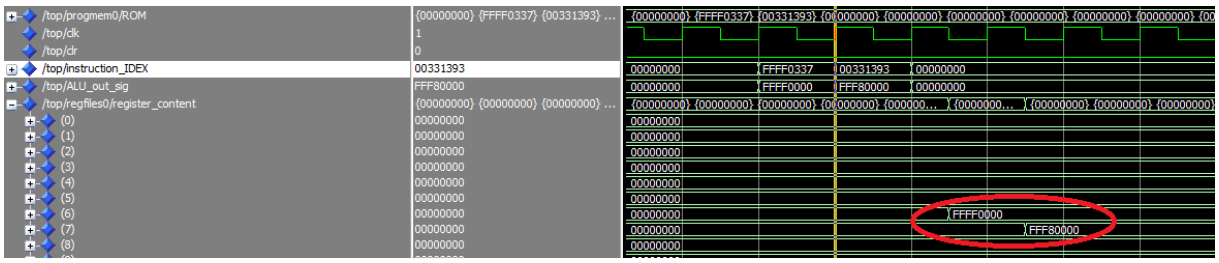
Gambar 4.40: Hasil pengujian instruksi ANDI.

Listing 4.29 merupakan program yang digunakan untuk menguji instruksi `ANDI`. Gambar 4.40 merupakan hasil pengujian *soft processor* dalam menjalankan program pengujian instruksi `ANDI` menggunakan *software* simulasi ModelSim. Untuk melakukan operasi AND terhadap suatu register asal dengan nilai *immediate*, dapat dimanfaatkan instruksi `ANDI`. Instruksi `andi x6 x5 31` dalam pengujian telah menyimpan hasil yang tepat dari operasi AND antara `x5` dan nilai *immediate* `31` ke dalam register `x6`.

25. SLLI

Listing 4.30: Program pengujian instruksi SLLI.

```
0:      ffff0337      lui  x6  0xffff0
4:      00331393      slli  x7  x6  3
```



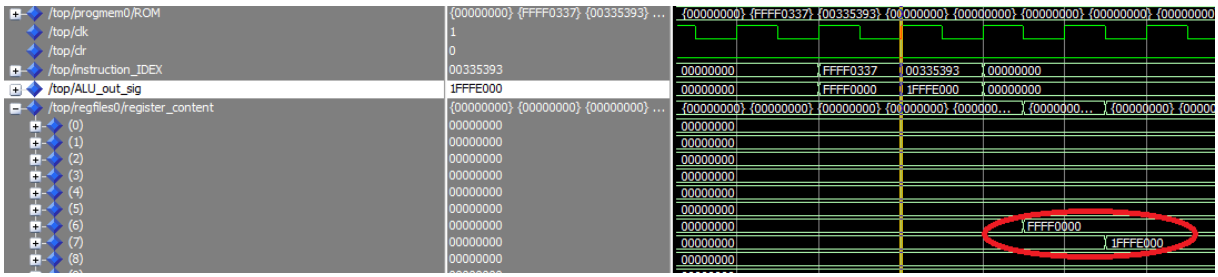
Gambar 4.41: Hasil pengujian instruksi SLLI.

Listing 4.30 merupakan program yang digunakan untuk menguji instruksi `SLLI`. Gambar 4.41 merupakan hasil pengujian *soft processor* dalam menjalankan program pengujian instruksi `SLLI` menggunakan *software* simulasi ModelSim. `SLLI` melakukan operasi *left logical shift* terhadap nilai dalam register asal dengan suatu nilai *immediate*. Pengujian instruksi `slli x7 x6 3` menghasilkan nilai register `x7` yang mengalami proses *left logical shift* sebanyak 3 kali. Hasilnya telah tersimpan dalam register tujuan `x7`.

26. SRLI

Listing 4.31: Program pengujian instruksi SRLI.

```
0:      ffff0337      lui x6 0xffff0
4:      00335393      srli x7 x6 3
```



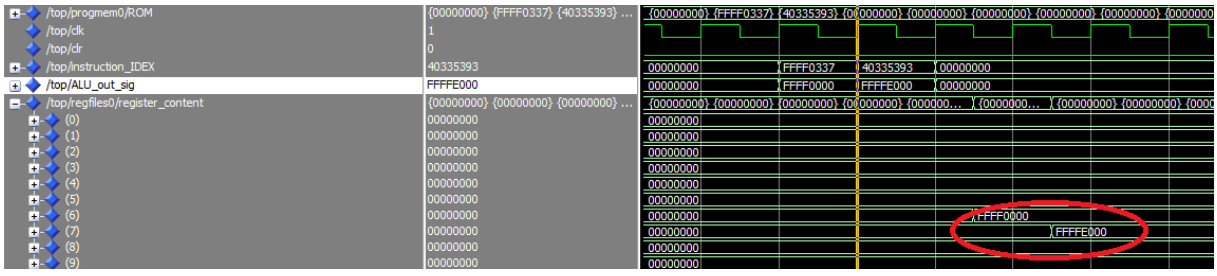
Gambar 4.42: Hasil pengujian instruksi SRLI.

Listing 4.31 merupakan program yang digunakan untuk menguji instruksi `SRLI`. Gambar 4.42 merupakan hasil pengujian *soft processor* dalam menjalankan program pengujian instruksi `SRLI` menggunakan *software* simulasi ModelSim. Selain *left shifting*, juga terdapat instruksi `SRLI` yang melakukan *right shifting* terhadap nilai dalam suatu register asal dengan nilai *immediate*. Instruksi `srli x7 x6 3` yang diujikan telah menyimpan nilai hasil *right shifting* register `x6` sebanyak tiga kali ke dalam register tujuan `x7`.

27. SRAI

Listing 4.32: Program pengujian instruksi SRAI.

```
0:      ffff0337      lui x6 0xffff0
4:      40335393      srai x7 x6 3
```



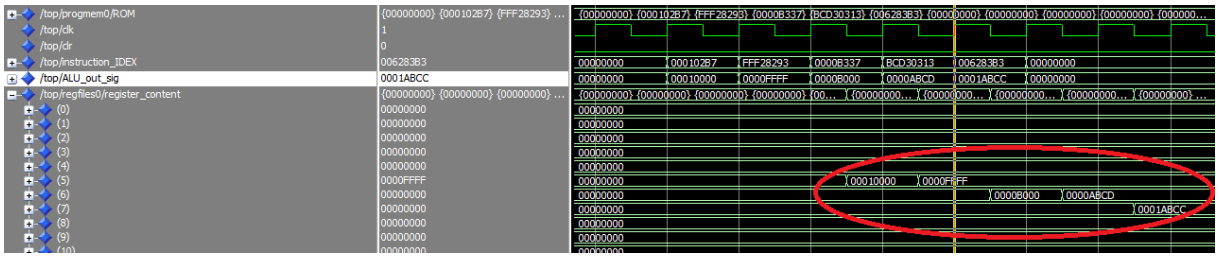
Gambar 4.43: Hasil pengujian instruksi SRAI.

Listing 4.32 merupakan program yang digunakan untuk menguji instruksi `SRAI`. Gambar 4.43 merupakan hasil pengujian *soft processor* dalam menjalankan program pengujian instruksi `SRAI` menggunakan *software* simulasi ModelSim. Dalam RV32I *ISA*, disediakan instruksi yang memungkinkan operasi *Right Arithmetic Shifting*. Berbeda dengan *logical shifting*, *arithmetic shifting* melakukan *sign-extension* terhadap data hasil *shifting*. Pada pengujian instruksi `srai x7 x6 3`, *sign-bit* dalam nilai *operand* register `x6` telah di-*sign-extend* pada saat operasi *right shifting* sebanyak tiga kali. Hasilnya disimpan dalam register tujuan `x7`.

28. ADD

Listing 4.33: Program pengujian instruksi ADD.

```
0:    000102b7    lui  x5  0x10
4:    fff28293    addi x5  x5  -1
8:    0000b337    lui  x6  0xb
c:    bcd30313    addi x6  x6  -1075
10:   006283b3    add  x7  x5  x6
```



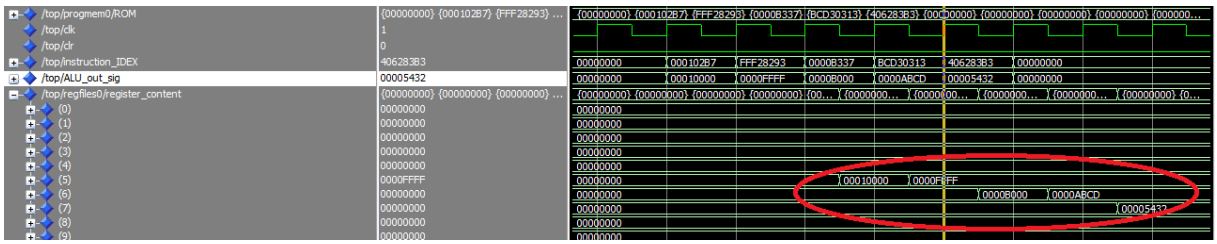
Gambar 4.44: Hasil pengujian instruksi ADD.

Listing 4.33 merupakan program yang digunakan untuk menguji instruksi `ADD`. Gambar 4.44 merupakan hasil pengujian *soft processor* dalam menjalankan program pengujian instruksi `ADD` menggunakan *software* simulasi ModelSim. Selain menjumlahkan suatu register asal dengan nilai *immediate*, dalam RISC-V *ISA* juga didefenisikan instruksi `ADD` yang menjumlahkan nilai dua *operand* yang berasal dari dua register dalam *register files*. Pada program pengujian, pertama-tama dimuat terlebih dahulu suatu nilai ke dalam register `x5` dan register `x6`. Selanjutnya, dilakukan operasi penjumlahan antara register-register tersebut menggunakan instruksi `add x7 x5 x6`. Hasilnya, seperti yang diinginkan, dituliskan dalam register `x7` pada tahap *Writeback*.

29. SUB

Listing 4.34: Program pengujian instruksi SUB.

```
0:    000102b7    lui  x5  0x10
4:    fff28293    addi x5  x5  -1
8:    0000b337    lui  x6  0xb
c:    bcd30313    addi x6  x6  -1075
10:   406283b3    sub  x7  x5  x6
```



Gambar 4.45: Hasil pengujian instruksi SUB.

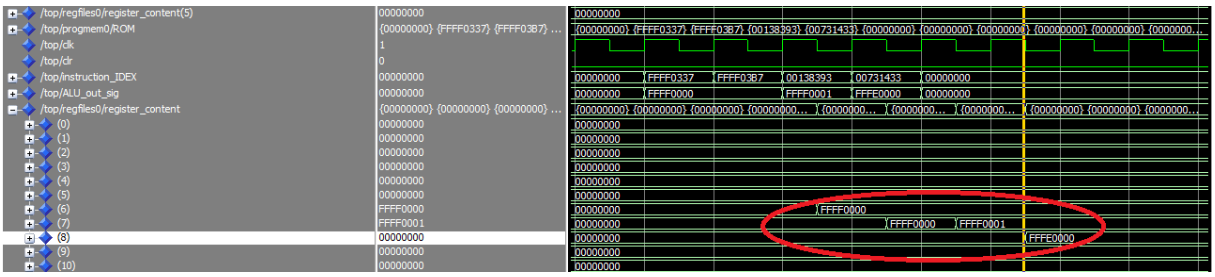
Listing 4.34 merupakan program yang digunakan untuk menguji instruksi `SUB`. Gambar 4.45 merupakan hasil pengujian *soft processor* dalam menjalankan program pengujian instruksi `SUB` menggunakan *software* simulasi ModelSim. Serupa dengan

instruksi `ADD`, instruksi `SUB` melakukan operasi pengurangan antara dua *operand* dari nilai register dalam *register files*. Dalam pengujian program, nilai dari register `x5` yang sebelumnya dimuat terlebih dahulu telah dikurangkan dengan nilai dalam register `x6`. Hasil yang tepat telah tersimpan dalam register tujuan `x7` setelah beberapa *clock cycle*.

30. SLL

Listing 4.35: Program pengujian instruksi SLL.

```
0:    ffff0337    lui x6 0xffff0
4:    ffff03b7    lui x7 0xffff0
8:    00138393    addi x7 x7 1
c:    00731433    sll x8 x6 x7
```



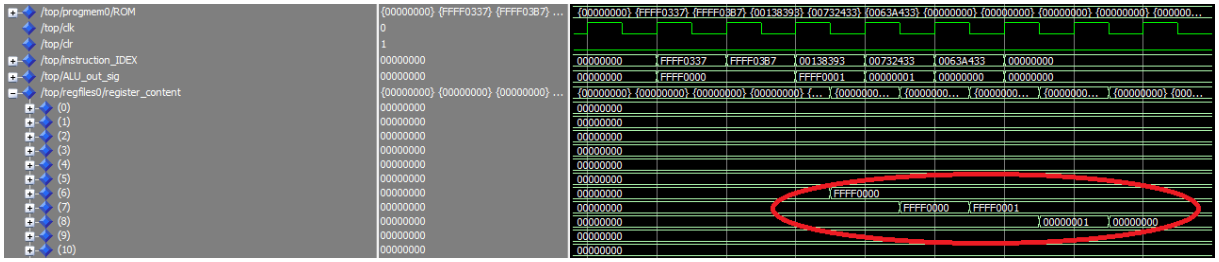
Gambar 4.46: Hasil pengujian instruksi SLL.

Listing 4.35 merupakan program yang digunakan untuk menguji instruksi `SLL`. Gambar 4.46 merupakan hasil pengujian *soft processor* dalam menjalankan program pengujian instruksi `SLL` menggunakan *software* simulasi ModelSim. `SLL` merupakan instruksi yang memuat nilai $\texttt{00000001}_{16}$ ke dalam register tujuan jika nilai *operand* pertama *ALU* lebih kecil daripada nilai *operand* kedua *ALU*. Kedua nilai *operand* tersebut berasal dari *register files*, tidak seperti pada instruksi `SLLI` yang merupakan varian *immediate* dari *Set Less Than*. Pada pengujian, *soft processor* telah berhasil menjalankan instruksi `SLL` dengan baik.

31. SLT

Listing 4.36: Program pengujian instruksi SLT.

```
0:    ffff0337    lui x6 0xffff0
4:    ffff03b7    lui x7 0xffff0
8:    00138393    addi x7 x7 1
c:    00732433    slt x8 x6 x7
10:   0063a433    slt x8 x7 x6
```



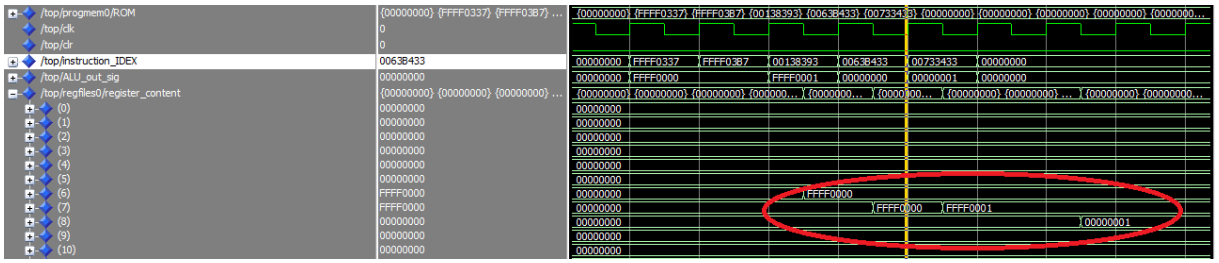
Gambar 4.47: Hasil pengujian instruksi SLT.

Listing 4.36 merupakan program yang digunakan untuk menguji instruksi `SLT`. Gambar 4.47 merupakan hasil pengujian *soft processor* dalam menjalankan program pengujian instruksi `SLT` menggunakan *software* simulasi ModelSim. `SLT` merupakan varian tipe-R dari *SLTI*. Sama seperti `SLTI`, `SLT` memuat nilai 00000001_{16} ke dalam register tujuan pada saat nilai *operand* pertama *ALU* lebih kecil daripada nilai *operand* yang kedua. Bedanya, kedua nilai *operand* berasal dari *register files*. Dalam pengujian, telah dimuat niai 00000001_{16} ke register `x8` pada saat *operand* pertama lebih kecil daripada *operand* kedua. Sedangkan, pada kasus sebaliknya, dituliskan nilai 00000000_{16}.

32. SLTU

Listing 4.37: Program pengujian instruksi SLTU.

```
0:      ffff0337      lui  x6 0xffff0
4:      ffff03b7      lui  x7 0xffff0
8:      00138393      addi  x7 x7 1
c:      0063b433      sltu  x8 x7 x6
10:     00733433      sltu  x8 x6 x7
```



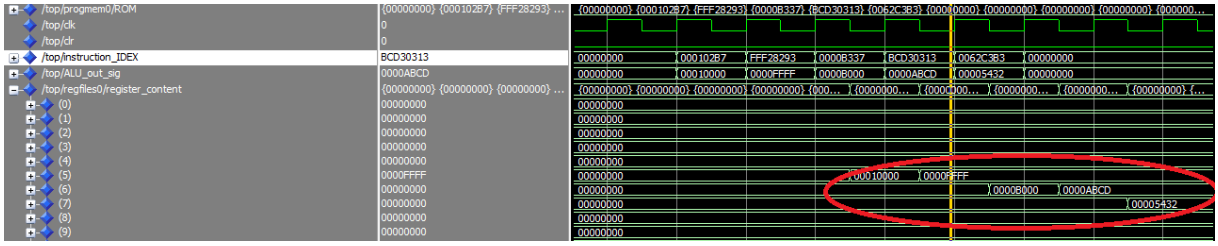
Gambar 4.48: Hasil pengujian instruksi SLTU.

Listing 4.37 merupakan program yang digunakan untuk menguji instruksi `SLTU`. Gambar 4.48 merupakan hasil pengujian *soft processor* dalam menjalankan program pengujian instruksi `SLTU` menggunakan *software* simulasi ModelSim. `SLTU` merupakan varian *unsigned dari* instruksi `SLT`. Sama seperti `SLT`, instruksi `SLTU` memuat nilai 00000001_{16} ke dalam register tujuan pada saat *operand* pertama lebih kecil dari *operand* kedua yang keduanya berasal dari *register files*. Bedanya, nilai *operand* yang digunakan dibaca sebagai suatu bilangan *unsigned*. Hasil pengujian berhasil membedakan operand sebagai nilai *unsigned*, dimana register `x7` lebih besar dari register `x6`. Hasil `SLTU` telah tersimpan dalam register `x8`.

33. XOR

Listing 4.38: Program pengujian instruksi XOR.

```
0:      000102b7      lui  x5  0x10
4:      fff28293      addi  x5  x5  -1
8:      0000b337      lui  x6  0xb
c:      bcd30313      addi  x6  x6  -1075
10:     0062c3b3      xor  x7  x5  x6
```



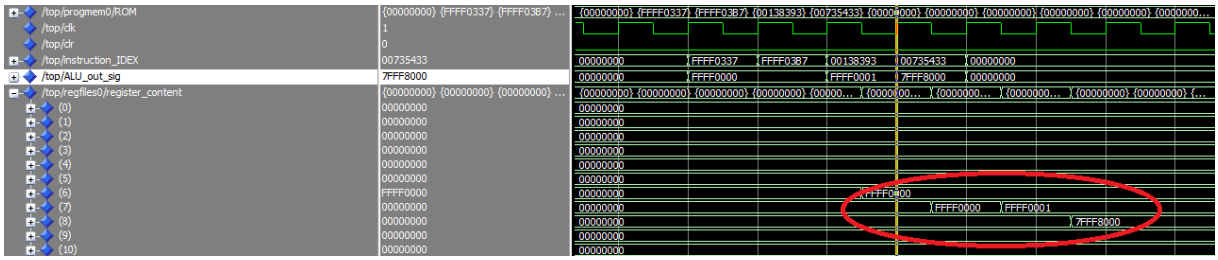
Gambar 4.49: Hasil pengujian instruksi XOR.

Listing 4.38 merupakan program yang digunakan untuk menguji instruksi `XOR`. Gambar 4.49 merupakan hasil pengujian *soft processor* dalam menjalankan program pengujian instruksi `XOR` menggunakan *software* simulasi ModelSim. Instruksi `XOR` melakukan operasi logika XOR terhadap *operand* pertama dan *operand* kedua *ALU* yang berasal dari *register files*. Pada pengujian program, operasi `xor x7 x5 x6` telah mampu menyimpan hasil operasi XOR antara `x5` dan `x6` ke dalam register tujuan `x7`.

34. SRL

Listing 4.39: Program pengujian instruksi SRL.

```
0:      ffff0337      lui  x6  0xffff0
4:      ffff03b7      lui  x7  0xffff0
8:      00138393      addi  x7  x7  1
c:      00735433      srl  x8  x6  x7
```



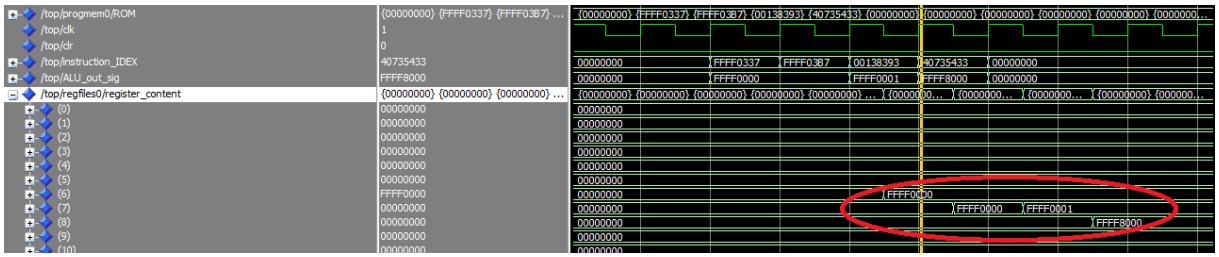
Gambar 4.50: Hasil pengujian instruksi SRL.

Listing 4.39 merupakan program yang digunakan untuk menguji instruksi `SRL`. Gambar 4.50 merupakan hasil pengujian *soft processor* dalam menjalankan program pengujian instruksi `SRL` menggunakan *software* simulasi ModelSim. Untuk melakukan operasi *right shifting logical*, digunakan sejumlah bit-bit terendah dari register asal *operand* kedua sebagai penanda berapa kali *shift* dilakukan. Dalam pengujian, instruksi `srl x8 x6 x7` telah berhasil melakukan insturksi *right shift* terhadap nilai dalam register `x6` dengan menggunakan bit-bit terendah `x7` sebagai acuan.

35. SRA

Listing 4.40: Program pengujian instruksi SRA.

```
0:      ffff0337      lui  x6 0xffff0
4:      ffff03b7      lui  x7 0xffff0
8:      00138393      addi x7 x7 1
c:      40735433      sra  x8 x6 x7
```



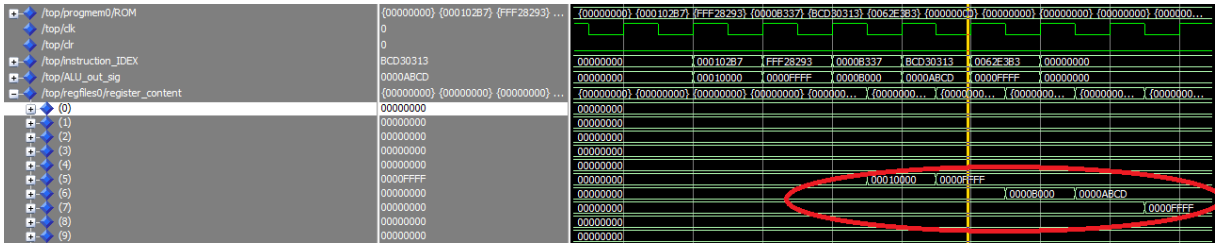
Gambar 4.51: Hasil pengujian instruksi SRA.

Listing 4.40 merupakan program yang digunakan untuk menguji instruksi `SRA`. Gambar 4.51 merupakan hasil pengujian *soft processor* dalam menjalankan program pengujian instruksi `SRA` menggunakan *software* simulasi ModelSim. Sama seperti instruksi `SRL`, instruksi `SRA` juga menggunakan *operand* yang nilainya berasal dari *register files*. Bedanya, instruksi `SRA` melakukan operasi *right rithmetic shifting* ketimbang *right logical shifting*. Program pengujian `sra x8 x6 x7` menunjukkan bahwa *soft processor* telah mampu melakukan *right arithmetic shifting* dengan cara melakukan *sign-extension* terhadap hasil *shifting* antara nilai register `x6` dan nilai bit-bit terendah register `x7`. Hasil yang tepat tersimpan dalam register `x8`.

36. OR

Listing 4.41: Program pengujian instruksi OR.

```
0:      000102b7      lui  x5 0x10
4:      fff28293      addi x5 x5 -1
8:      0000b337      lui  x6 0xb
c:      bcd30313      addi x6 x6 -1075
10:     0062e3b3      or   x7 x5 x6
```



Gambar 4.52: Hasil pengujian instruksi OR.

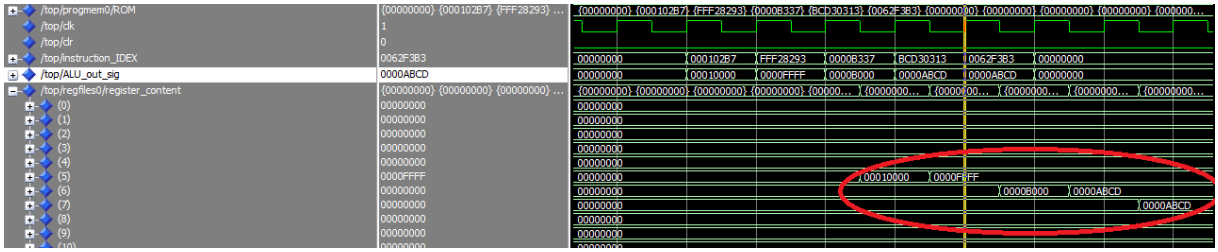
Listing 4.41 merupakan program yang digunakan untuk menguji instruksi `OR`. Gambar 4.52 merupakan hasil pengujian *soft processor* dalam menjalankan program pengujian instruksi `OR` menggunakan *software* simulasi ModelSim. Instruksi `OR` melakukan operasi logika OR terhadap nilai *operand* yang berasal dari register-register

dalam *Register Files*. Pada program pengujian, suatu nilai dimuat ke dalam register `x5` dan `x6` menggunakan instruksi *Load Upper Immediate* dan *Add Immediate*. Kemudian, dilakukan operasi *OR* terhadap kedua register tersebut menggunakan instruksi `or x7 x5 x6`. Hasil yang sudah sesuai dengan spesifikasi RV32 disimpan ke dalam register destinasi `x7`.

37. AND

Listing 4.42: Program pengujian instruksi AND.

```
0:    000102b7    lui  x5 0x10
4:    fff28293    addi x5 x5 -1
8:    0000b337    lui  x6 0xb
c:    bcd30313    addi x6 x6 -1075
10:   0062f3b3    and  x7 x5 x6
```



Gambar 4.53: Hasil pengujian instruksi AND.

Listing 4.42 merupakan program yang digunakan untuk menguji instruksi `AND`. Gambar 4.53 merupakan hasil pengujian *soft processor* dalam menjalankan program pengujian instruksi `AND` menggunakan *software* simulasi ModelSim. Untuk melakukan operasi AND terhadap dua *operand* yang keduanya berasal dari *Register Files*, digunakan instruksi tipe-R `AND`. Pada program pengujian, dilakukan operasi logika AND terhadap nilai register `x5` dan `x6` dan disimpan ke dalam register tujuan `x7` menggunakan instruksi `and x7 x5 x6`. Hasil yang dicapai sudah sesuai dengan spesifikasi RV32 *ISA*.

[Halaman ini sengaja dikosongkan]

BAB V
PENUTUP

5.1 Kesimpulan

Berdasarkan hasil pengujian *soft processor* yang didapatkan, penulis menyimpulkan sejumlah hal:

1. *Soft processor* yang diajukan mampu menjalankan semua instruksi dalam RISC-V RV32I *Instruction Architecture* kecuali `FENCE`, `CSSR`, dan `ECALL`. *Glitch* yang terjadi pada *soft processor core* Maestro telah diperbaiki dalam penelitian ini. *Core* ini tidak mengimplementasikan instruksi-instruksi tersebut dikarenakan *soft processor* tidak ditujukan untuk mendukung kapabilitas *multithreading*.
2. *Soft Processor* yang diajukan membutuhkan 2.115 *LUT*, 558 *flip-flop*, dan 67.608 *memory bits* untuk disintesis ke dalam *FPGA* Cyclone IV EP4CE6E22C.
3. *Soft Processor* yang diajukan mampu berjalan dengan frekuensi maksimal 62.95 MHz dalam *FPGA* Cyclone IV EP4CE6E22C. Frekuensi maksimal ini berjalan 37% lebih cepat dari desain prosesor *single-cycle* serupa yang dirancang sebelumnya.
4. *Critical path* atau jalur terpanjang dari *datapath* desain prosesor meliputi jalur yang menghubungkan register *pipeline* `MEM/WB`, multiplekser `MEM/WB`, multiplekser *ALU*, *ALU*, *ALU branch detector*, *jump/branch detector*, multiplekser *Program Counter*, dan *Program Counter*. Jalur kritis ini mempengaruhi frekuensi maksimal yang dapat dijalankan *soft processor*.

5.2 Saran

Sedangkan, dari proses pengembangan *soft processor* yang telah berlangsung, penulis memiliki sejumlah masukan:

1. Dapat dikembangkan suatu sistem *branch prediction* yang dapat meningkatkan frekuensi maksimal dan *instruction per cycle* dari prosesor. Prosesor yang diajukan masih mendeteksi *branch taken* dan *jump* menggunakan *ALU*.
2. Frekuensi maksimal dapat ditingkatkan dengan mengoptimasi *critical path* dari *soft processor*. Optimasi ini dilakukan dengan cara menyeimbangkan logika kombinasional dalam arsitektur prosesor.
3. Untuk memastikan *soft processor* berjalan sesuai dengan spesifikasi RISC-V, dapat dilakukan *formal verification* sesuai dengan himbauan RISC-V *foundation*. Verifikasi logika prosesor pada proyek ini masih dilakukan secara manual dengan menguji instruksi, *hazard*, dan tes pengujian satu per satu.
4. Agar prosesor dapat berkomunikasi dengan perangkat lain, perlu dikembangkan sistem *input/output*. Prosesor yang dikembangkan masih menggunakan keluaran berupa *display seven segment* saja. Prosesor dapat menggunakan protokol *Universal Asynchronous Receiver Transmitter* (*UART*) untuk menerima dan menyalurkan *data* secara *serial*.

[Halaman ini sengaja dikosongkan]

DAFTAR PUSTAKA

Aaron Elson, P. (2022). Synthesizable single-cycle fpga soft processor core.

Asanović, K. and Patterson, D. A. (2014). Instruction sets should be free: The case for risc-v. Technical Report UCB/EECS-2014-146, EECS Department, University of California, Berkeley.

Asicnorth (2021). Asic vs. fpga: What's the difference?: Asic north inc.

Dahad, N. (2018). Arm offers lower cost cortex-a5 license.

Krolikoski, S. (2011). Evolution of eda standards worldwide. *IEEE Design Test of Computers*, 28(1):72–75.

McLoughlin, I. (2018). *Computer Systems: An Embedded Approach*. McGraw-Hill Education.

Patterson, D. A. and Hennessy, J. L. (2017). *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.

Poorhosseini, M., Nebel, W., and Grüttner, K. (2020). A compiler comparison in the risc-v ecosystem.

Qui, N. M., Lin, C. H., and Chen, P. (2020). Design and implementation of a 256-bit risc-v-based dynamically scheduled very long instruction word on fpga. *IEEE Access*, 8:172996–173007.

Rafique, Z. (2019). Merledu/risc-v-single-cycle-core-logisim: This repository is for risc-v single cycle core.

Singh, R. and Rajawat, A. (2013). A review of fpga-based design methodologies for efficient hardware area estimation. *IOSR Journal of Computer Engineering*, 13:01–06.

Tong, J. G., Anderson, I. D. L., and Khalid, M. A. S. (2006). Soft-core processors for embedded systems. In *2006 International Conference on Microelectronics*, pages 170–173.

Vitor Rafael Chrisóstomo, J. (2018). Entendendo a necessidade de uma isa aberta e implementação de um núcleo risc-v. *Github*.

Waterman, A., Lee, Y., Patterson, D. A., and Asanovi, K. (2014). *The RISC-V Instruction Set Manual. Volume 1: User-Level ISA, Version 2.0*.

[Halaman ini sengaja dikosongkan]

BIOGRAFI PENULIS

Aaron Elson Phangestu, lahir pada 10 Mei 2000 di DKI Jakarta, merupakan mahasiswa tingkat akhir di Departemen Teknik Komputer Institut Teknologi Sepuluh Nopember. Merupakan anak pertama dari dua saudara. Lulus dari SMPK 2 BPK PENABUR pada tahun 2015 dan dari SMAK 1 BPK PENABUR pada tahun 2018. Penulis pernah menjadi panitia tim pembuatan soal dalam acara tahunan MAGE 5 & 6. Selama kuliah, penulis tertarik dalam bidang desain digital, FPGA, dan prosesor lunak. Bagi pembaca yang memiliki kritik, saran, atau pertanyaan, dapat menghubungi penulis melalui e-mail aaronelsonp@gmail.com.

Ditetapkan di Surabaya
REKTOR INSTITUT TEKNOLOGI
SEPULUH NOPEMBER,

MOCHAMAD ASHARI
NIP 196510121990031003

[Halaman ini sengaja dikosongkan]