https://doi.org/10.24123/saintek.v5i1.6306

Original Research

Implementasi *Hazard Detection* dan *Data Forwarding* pada *Soft-core* Prosesor Sederhana

Henry Hermawan [1]*

[1] Jurusan Teknik Elektro, Fakultas Teknik, Universitas Surabaya, Surabaya, Indonesia
* corresponding author: henryhermawan@staff.ubaya.ac.id

Abstract—*This paper presents the implementation of hazard detection and data forwarding in the pipeline concept used in developing TEUS-1, a simple, 16-bit, soft-core processor. This processor is planned to be used for the learning process related to examples of processor design for undergraduate students. The pipeline implemented on TEUS-1 is 5 stages. In initial testing, 10 data hazards were found due to the existence of data dependencies between sequential instructions. In order to eliminate these data hazards, an interlocking algorithm and a data forwarding algorithm has been added to the instruction decoding section and to the execution unit section, respectively. With the addition of this algorithm, all data hazards found during initial testing can be eliminated.*

Keywords: *hazard detection, data forwarding, interlocking, data hazard, pipeline*

Abstrak—Makalah ini mempresentasikan tentang implementasi dari hazard detection dan data forwarding pada konsep pipeline yang digunakan dalam pengembangan soft-core prosesor sederhana, yaitu TEUS-1. Prosesor ini merupakan prosesor 16 bits yang rencananya akan digunakan untuk proses pembelajaran terkait dengan contoh desain prosesor untuk mahasiswa program Sarjana. Pipeline yang diimplementasikan pada TEUS-1 adalah 5 stages. Dalam pengujian awal, ditemukan adanya 10 data hazard akibat dari adanya data dependencies antar instruksi yang berurutan. Untuk menghilangkan data hazard ini, telah ditambahkan algoritma interlocking pada bagian instruction decoding dan algoritma data forwarding pada bagian execution unit. Dengan adanya penambahan algoritma ini, seluruh data hazard yang ditemukan pada saat pengujian awal dapat dihilangkan.

Kata kunci: *hazard detection, data forwarding, interlocking, data hazard, pipeline*

PENDAHULUAN

Dalam kurun dekade terakhir, desain *soft-core processor* telah berkembang sangat pesat. Hal ini dapat teramati pada salah satu situs internet, http://opencores.org, yang khusus didedikasikan untuk pengembangan OpenRISC *soft-core processor* dan *open-source hardware IP-cores* lainnya (*OpenCores*, n.d.). Selain itu, masih banyak lagi jenis *soft-core processor* yang layak untuk digunakan pada produk komersial seperti MicroBlaze, Nios II, ARM Cortex-M1, LatticeMico32, LEON3, S1-Core, dan OpenSPARC T1 (Hermawan, 2012). Hanya saja, prosesor-prosesor tersebut ditujukan untuk aplikasi komersial, bukan untuk pendidikan terkait teknologi prosesor, karena kompleksitasnya yang sulit dipelajari oleh pembelajar tingkat awal dan menengah.

Selain beberapa *soft-core processor* yang memang ditujukan untuk aplikasi komersial (Hermawan, 2012; Tong et al., 2006), ada beberapa *soft-core processor* yang didesain untuk tujuan penelitian maupun pembelajaran, dan ditujukan atau dioptimalkan untuk sasaran platform FPGA. Bahkan beberapa di antaranya bersifat terbuka (*open source*). MB-Lite (Kranenburg & van Leuken, 2010) dan The Secret Blaze (Barthe et al., 2011) merupakan *soft-core processor* yang bersifat terbuka. Kedua prosesor ini menggunakan *Instruction Set Architecture* (ISA) dari prosesor komersial MicroBlaze (*AMD MicroBlaze Processor: A Flexible and Efficient Soft Processor*, 2023) dan dapat dikonfigurasi ulang sesuai dengan kebutuhan aplikasi. Selain kedua prosesor tersebut, telah didesain Tinuso (Schleuniger et al., 2012) yang secara khusus dioptimalkan untuk implementasi pada FPGA. Cara yang digunakan pada Tinuso adalah *superpipelining* yang dapat meningkatkan frekuensi kerja sistem dan mengimplementasikan teknik *predicated instructions* yang dapat mengurangi *pipeline stalls* karena adanya alur percabangan pada program. Octavo (LaForest & Steffan, 2012) dan iDEA (Cheah et al., 2014; Cheah, Fahmy, & Maskell, 2012; Cheah, Fahmy, Maskell, et al., 2012) merupakan prosesor dalam tahap penelitian yang didesain untuk mengoptimalkan

https://doi.org/10.24123/saintek.v5i1.6306

pemanfaatan *hard macros* pada FPGA. *Hard macros*, misalnya DSP *Blocks* dan BRAM, telah banyak ditanamkan pada FPGA terkini. Penggunaan *hard macros* ini dapat meningkatkan performa sistem dan mengurangi penggunaan *Look-up Tables* (LUTs), sehingga dapat dihasilkan suatu sistem yang kecil tetapi mempunyai performa yang tinggi.

Patterson (Patterson & Hennessy, 2020a, 2020b), Baer (Jean-Loup Baer, 2009), Shen (Shen & Lipasti, 2013), dan Silc (Silc et al., 1999) memberikan contoh detil tentang desain *pipelined processor* yang dijadikan referensi untuk mata kuliah tentang desain prosesor. Contoh dari Patterson merupakan prosesor MIPS dan RISC-V yang memiliki 5 *pipeline stages*, dilengkapi dengan *hazard detection unit* dan *data forwarding* untuk menangani permasalahan *data hazard* yang seringkali muncul dalam pengeksekusian instruksi oleh prosesor. Baer, Shen, dan Silc lebih dalam lagi menjelaskan tentang *data hazard* yang disebabkan oleh *data dependence*, *structural hazard* (*resources conflict*), dan *control hazard* beserta dengan detil solusinya seperti *interlocking*, *forwarding*, kombinasi antara *interlocking* dengan *forwarding*, *resources replication*, *delayed jump/branch technique*, serta *branch prediction technique*. Namun, referensi-referensi tersebut tidak memberikan langkah-langkah desain yang konkret dan detil terkait implementasi *hazard detection unit* dan *data forwarding*. Namun, contoh-contoh yang diulas pada referensi-referensi tersebut akan dijadikan referensi utama dalam mendesain teknologi *pipeline* dari TEUS-1 yang nantinya akan dijabarkan langkah-langkah desainnya.

Dalam bukunya, Nurmi (2007) mengompilasi artikel-artikel tentang proses desain prosesor dari awal hingga dapat diimplementasikan pada ASIC dan FPGA. Dua artikel membahas tentang alur desain prosesor mulai dari tahap pengidentifikasian kebutuhan fitur dasar, penentuan awal *instruction set*, pengidentifikasian lanjutan untuk kebutuhan fitur tambahan yang disesuaikan dengan aplikasi dari prosesor tersebut, finalisasi *instruction set*, sampai dengan tahap HDL *synthesis* beserta dengan contoh desain Coffee Processor mulai dari tahap awal sampai dengan siap untuk diimplementasikan pada FPGA. Alur desain tersebut akan diadopsi pada penelitian ini.

Untuk pembelajaran tentang desain prosesor, terdapat beberapa prosesor yang cukup sederhana untuk dipelajari. Prosesor-prosesor tersebut mulai dari yang berarsitektur 8 bits, contohnya Natalius (Figueroa, 2012), Dracon (Parrilla et al., 2022), sampai dengan yang berarsitektur 16 bits dan 32 bits seperti DLX, MIPS16, MIPS32, dan RISC-V (Patterson & Hennessy, 2020a, 2020b). Prosesor-prosesor ini merupakan *pipeline processor* yang sederhana karena tidak mempunyai antarmuka untuk memori eksternal, dengan kata lain, prosesor ini menggunakan memori internal. MIPS16 akan menjadi referensi utama untuk pengembangan TEUS-1 agar dapat dipelajari dengan mudah saat digunakan pada proses pembelajaran mahasiswa program Sarjana.

METODE

Tahap awal yang dilakukan adalah melakukan studi terhadap beberapa *pipelined processor* yang telah didesain dalam kurun dekade terakhir. Prosesor yang dipilih adalah prosesor hasil penelitian oleh akademisi dari perguruan tinggi lain dan prosesor yang merupakan hasil HDL (*Hardware Description Language*) *synthesis* seperti MIPS32, MIPS16 (Patterson & Hennessy, 2020a, 2020b), dan Natalius *processor* (Figueroa, 2012). Langkah pertama ini dimaksudkan untuk melakukan survei tentang apa yang telah dikerjakan oleh kolega serta membantu dalam melakukan identifikasi kebutuhan fitur dasar untuk TEUS-1. Selain itu, studi ini juga untuk mempelajari solusi untuk *data hazard* menggunakan fitur *hazard detection* dan *data forwarding* yang sangat jarang ditemukan contoh implementasinya yang cukup detil.

Selanjutnya, hasil dari studi banding dijadikan sebagai masukan awal untuk penyusunan konsep atau blok desain. Pada tahap ini, langkah pertama adalah melakukan identifikasi kebutuhan fitur dasar bagi TEUS-1. Fitur dasar ini meliputi “kedalaman” *pipeline* atau *pipeline stages*, *instruction & data memory*, jenis *registers*, dan sebagainya. Setelah

https://doi.org/10.24123/saintek.v5i1.6306

identifikasi kebutuhan fitur dasar selesai dikerjakan, langkah selanjutnya adalah menyusun prototipe dari *instruction set* berdasarkan hasil identifikasi fitur dasar. Biasanya, *instruction set* ini berisi instruksi-instruksi dasar dari prosesor. Berikutnya, akan dilakukan identifikasi kebutuhan fitur tambahan yang disesuaikan dengan "*environment*", maksudnya, disesuaikan dengan aplikasi yang akan diterapkan pada TEUS-1. Untuk prototipe TEUS-1 ini, aplikasi yang dipilih adalah aplikasi prosesor untuk pembelajaran mata kuliah Arsitektur Mikroprosesor dan dipersiapkan juga untuk dikembangkan dengan sasaran aplikasi penelitian bersama di jurusan Teknik Elektro. Berdasarkan identifikasi kebutuhan fitur tambahan, dilakukan penambahan instruksi dan finalisasi *instruction set* yang meliputi penentuan *instruction format*, *coding*, penggunaan *registers*, dan lain sebagainya. Langkah terakhir dari tahap kedua ini adalah menyusun *architecture organization* dari TEUS-1 dengan *managing complexity*, menyusun *datapath structure*, menentukan *control strategy* dan *control flow*, menambahkan fitur *hazard detection* dan *data forwarding*, serta melakukan *design partitioning* ke dalam modul-modul yang lebih sederhana sehingga memudahkan HDL *synthesis*.

Tahap ketiga adalah HDL *synthesis*. Tahap ini merupakan tahap untuk mendeskripsikan *hardware* menggunakan bahasa deskripsi *hardware* (*Hardware Description Language*, disingkat dengan HDL) berdasarkan *architecture organization* yang telah disusun pada tahap kedua. Pendekatan yang diambil adalah pendekatan *bottom-up*, dengan kata lain, sistesis dilakukan mulai dari modul-modul yang paling sederhana dulu, diverifikasi, baru kemudian diintegrasikan dan diverifikasi hasil integrasinya. Verifikasi masing-masing modul dan integrasi per bagian modul dilakukan dengan cara *gate-level simulation* menggunakan *testbench files*. Setelah modul-modul terintegrasi secara keseluruhan, akan dilakukan validasi desain dengan cara *gate-level simulation* menggunakan *testbench file* dan *test program*.

HASIL DAN BAHASAN

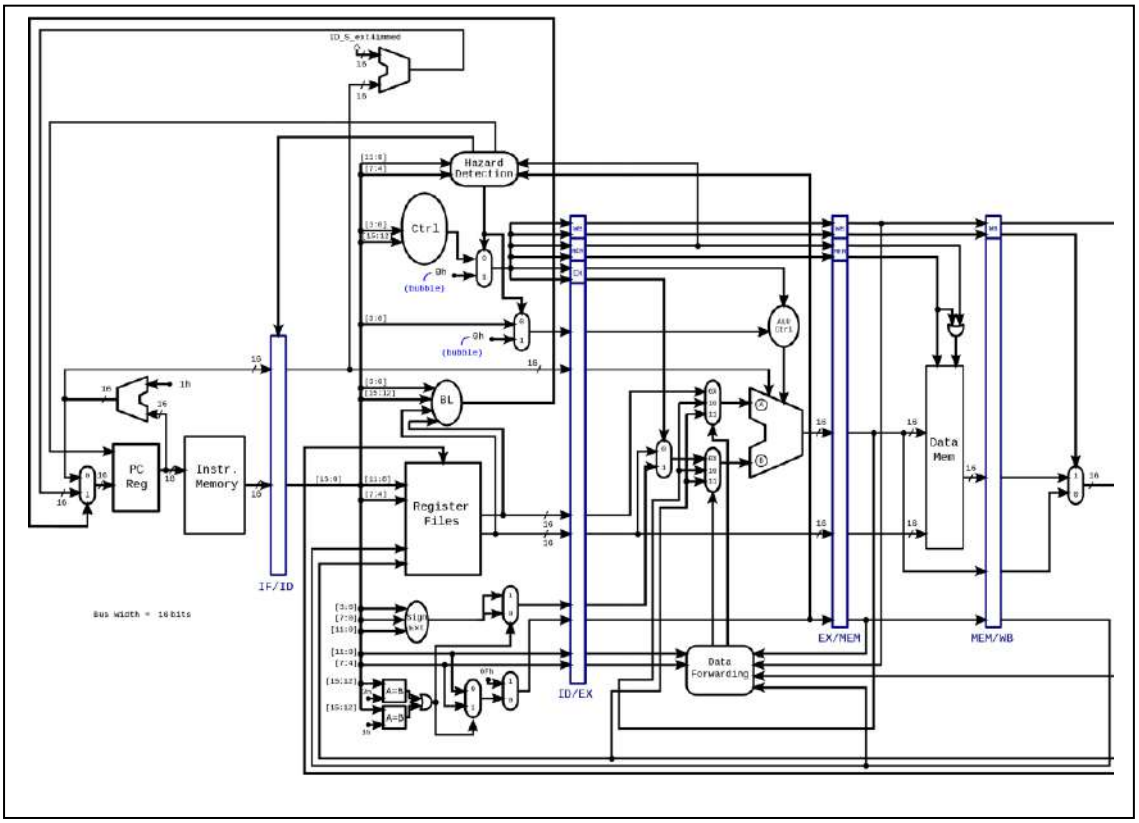
Prosesor TEUS-1 merupakan prosesor kecil, yang menangani data dalam format 16 bits menggunakan memori untuk instruksi dan data yang diposisikan pada internal prosesor. Jadi, dalam desain awal dari prosesor TEUS-1 ini, dipilih untuk tidak mengimplementasikan *memory controller* yang digunakan untuk mengakses memori eksternal. Pemilihan ini didasarkan dari hasil studi dari beberapa prosesor yang menunjukkan kesederhanaan desain jika tanpa *memory controller* sehingga TEUS-1 lebih mudah untuk dijadikan contoh desain prosesor.

Gambar 1 menunjukkan diagram internal dari prosesor TEUS-1 dalam bentuk yang disederhanakan. Prosesor TEUS-1 ini menggunakan konsep *pipeline* yang mempunyai 5 *stages* untuk setiap siklus eksekusi instruksi, yaitu: IF Stage, ID Stage, EX Stage, MEM Stage, dan WB Stage. Implementasi *5-stage pipeline* ini merupakan implementasi sederhana dan relatif mudah dipelajari dibandingkan dengan implementasi *pipeline* yang lebih dari 5 *stages* dan cukup kompleks bila dibandingkan dengan *3-stage pipeline*. Pada tahap IF Stage, TEUS-1 melakukan *instruction fetch*, yaitu mengambil instruksi yang disimpan pada *instruction memory* untuk ditransfer ke ID Stage dan bagian *operation code* (*op-code*) dari instruksi tersebut "diterjemahkan" atau di-*decoded* menjadi satu kumpulan sinyal kontrol. Sinyal kontrol ini akan digunakan untuk mengontrol proses yang akan dikerjakan pada *stage* selanjutnya. Selain "menerjemahkan" *op-code*, TEUS-1 juga melakukan proses pengambilan data dari *register* dan juga melakukan penulisan data ke *register*. Data yang diambil dari *register* ini merupakan *operand* dari instruksi, atau dengan kata lain, data tersebut yang akan diproses pada *stage* selanjutnya. Pemrosesan data dikerjakan pada EX Stage. *Stage* ini memiliki Arithmetic and Logic Unit (ALU) yang akan memproses data dengan operasi aritmatika dan logika. Hasil dari pemrosesan data ini akan disimpan pada *data memory* pada MEM Stage atau ditransfer ke *register* untuk disimpan pada WB Stage.

TEUS-1 mempunyai 16 *general purpose registers* yang diberi nama R0 – R15, jumlah *general purpose registers* yang umum untuk prosesor sederhana. Format/lebar data dari *register* ini adalah 16 bits. Selain berfungsi sebagai *general purpose registers*, R14 dan R15 juga

https://doi.org/10.24123/saintek.v5i1.6306

berfungsi sebagai *stack pointer register* pada operasi yang melibatkan *stack memory* dan *return address call register* pada proses pemanggilan suatu fungsi.



Gambar 1. Diagram internal TEUS-1 (yang disederhanakan).

TEUS-1 mempunyai format instruksi yang mengadopsi format instruksi dari prosesor MIPS (Patterson & Hennessy, 2020a), yang cukup sederhana untuk dipelajari. Panjang instruksi dari TEUS-1 adalah 16 *bits*, yang dikelompokkan menjadi 3 tipe format, yaitu: R-format, I-format, dan J-format (Tabel 1). R-format merupakan format dari hampir semua instruksi TEUS-1. Jenis instruksi yang menggunakan format ini adalah instruksi-instruksi aritmatika, logika, *shift*, dan *rotate*. Instruksi yang membutuhkan *immediate data* akan menggunakan I-format. J-format digunakan untuk instruksi-instruksi yang terkait dengan percabangan (*branching*) dan perulangan (*looping*).

Tabel 1
Format Instruksi TEUS-1

Field	4 bits	4 bits	4 bits	4 bits
R-format	op	Rd	rs/shamt	Funct/Short Immediate/Short Displacement
I-format	op	Rd	Immediate (8bits)	
J-format	op	Target address (12 bits)		

Sinyal kontrol yang dibutuhkan sebagai hasil *decoding* dari instruksi-instruksi yang direncanakan, adalah sebagai berikut.

https://doi.org/10.24123/saintek.v5i1.6306

- *RegWrite*
  Sinyal kontrol *RegWrite* digunakan untuk memberikan sinyal bahwa hasil dari operasi yang dilakukan akan disimpan/dituliskan kembali ke suatu *register*.
- *MemToReg*
  Sinyal kontrol ini digunakan untuk memilih *data source* yang akan dituliskan ke suatu *register*. Pilihan *data source* yang tersedia adalah *data memory* dan ALU.
- *Branch*
  Sinyal kontrol *Branch* digunakan untuk memberikan sinyal bahwa operasi yang akan dikerjakan adalah operasi percabangan dari suatu instruksi percabangan.
- *MemRead*
  Sinyal kontrol *MemRead* merupakan sinyal kontrol yang mengindikasikan operasi pembacaan data (*data read*) dari *data memory*.
- *MemWrite*
  Sinyal kontrol *MemWrite* adalah sinyal kontrol yang mengindikasikan operasi penulisan data (*data write*) ke *data memory*.
- *RegDst*
  Sinyal kontrol *RegDst* digunakan untuk memilih *destination register* yang akan menjadi tujuan penyimpanan data.
- *ALUSrc*
  Sinyal kontrol *ALUSrc* digunakan untuk memilih *data source* yang akan menjadi *the second ALU operand*.
- *ALUOp*
  ALUOp merupakan *encoded op-code* yang di-*decoded* oleh *ALU Control Unit* untuk menentukan fungsi yang akan diaktifkan untuk operasi yang akan dikerjakan di ALU.

Tabel 2
Program Sederhana Analisis Data Hazard

Alamat Memori	Machine Instruction	Assembly	Alamat Memori	Machine Instruction	Assembly
00	0000	nop	1A	9303	bne $3, $0, #3
01	4101	lwi $1, #1	1B	0000	nop
02	4202	lwi $2, #2	1C	0121	add $1, $2
03	43FF	lwi $3, #-1	1D	021A	nand $2, $1
04	44A3	lwi $4, #0xA3	1E	0541	add $5, $4
05	0000	nop	1F	013C	xor $1, $3
06	0000	nop	20	0632	sub $6, $3
07	54F8	lui $4, #0xF8	21	0000	nop
08	C025	jal #25	22	8007	beq $0,$0,#7
09	0000	nop	23	0000	nop
0A	0535	mov $5, $3	24	0000	nop
0B	0411	add $4, $1	25	0241	add $2, $4
0C	6505	addi $5, #5	26	47BC	lwi $7, #0xBC
0D	0548	and $5, $4	27	570A	lui $7, #0xA
0E	3100	sw $1, #0($0)	28	00FE	jr $15
0F	3201	sw $2, #1($0)	29	0000	nop
10	3302	sw $3, #2($0)	2A	412B	lwi $1, #0x2B
11	3312	sw $3, #2($1)	2B	0405	mov $4, $0
12	6202	add $2, #2	2C	0476	mfu $4, $7
13	2042	lw $4, #4($0)	2D	0477	mtu $4, $7
14	0411	add $4, $1	2E	1320	sll $3, #2
15	0000	nop	2F	1332	sla $3, #3
16	2161	lw $6, #1($1)	30	001F	jlr $1
17	0569	or $5, $6	31	0000	nop
18	0621	add $6, $2			
19	0000	nop			

https://doi.org/10.24123/saintek.v5i1.6306

Analisis Data Hazard

Sayangnya, pada literatur yang dipelajari, tidak dicantumkan contoh implementasi *data hazard*. Yang tersedia hanyalah analisis *data hazard* secara teoretis. Untuk itu, saat melakukan analisis *data hazard* dalam implementasi *pipeline* pada TEUS-1, disusun suatu program sederhana dengan berbagai kombinasi *data hazard* yang mungkin terjadi (Silc et al., 1999). Program sederhana tersebut dapat dilihat pada Tabel 2.

Dengan program sederhana tersebut, telah ditemukan ada 10 *data hazard* yang akan muncul. Gambar 2 menunjukkan temuan *data hazard* #1, #2, dan #3 yang muncul akibat kombinasi *data dependencies* yang mirip seperti kombinasi instruksi-instruksi yang ditandai pada gambar. *Data hazard* yang pertama, instruksi ADDI $5,#5 membutuhkan *updated operand* yang disimpan pada *register* R5, di mana *updated operand* tersebut berasal dari instruksi sebelumnya, yaitu instruksi MOV $5,$3. Supaya nilai "baru" dari R5 dapat digunakan oleh instruksi ADDI $5,#5, dibutuhkan solusi *data forwarding*. Temuan berikutnya, *data hazard* #2, adalah *data dependency* dari instruksi AND $5,$4. Intruksi tersebut membutuhkan *updated operand* pada *register* R4 yang berasal dari instruksi ADD $4,$1 saat memasuki *EX Stage*. Kenyataannya, saat instruksi AND $5,$4 memasuki *EX Stage*, nilai yang tersimpan pada *register* R4 belum ter-*updated* sehingga perlu solusi *data forwarding* untuk mengatasi *data hazard* ini. *Data hazard* ketiga juga muncul karena instruksi AND $5,$4 membutuhkan *updated operand* pada *register* R5. Saat memasuki *EX stage*, nilai R5 belum ter-*updated* karena nilai baru hasil dari eksekusi ADDI $5,#5 belum dituliskan ke *register* R5. Solusi untuk *data hazard* #3 ini juga *data forwarding*.

Gambar 3 menunjukkan temuan *data hazard* #4 dan #5. *Data hazard* #4 ditemukan untuk kombinasi *data dependency* yang menyerupai *data dependency* dari instruksi ADD $4,$1 dengan LW $4,#2($0). Seharusnya, instruksi LW $4,#2($0) menyediakan *updated data* yang disimpan pada *register* R4 untuk instruksi ADD $4,$1 saat instruksi tersebut memasuki *EX stage*. Namun, pada *cycle* tersebut, nilai yang baru belum tersimpan pada *register* R4 saat dibutuhkan; nilai baru tersebut akan tersedia pada *cycle* berikutnya. Karena itu, solusi untuk *data hazard* dengan kombinasi seperti ini adalah *interlocking solution* (*stall the pipeline*) yang dilakukan oleh *hazard detection unit*. Walaupun telah dilakukan *interlocking solution*, ternyata nilai baru tersebut masih tetap belum ter-*updated* pada *register* R4 (*data hazard* #5). Nilai baru tersebut baru akan ter-*updated* pada R4 setelah *cycle* berikutnya, sehingga, untuk mengatasi *data hazard* #5 ini, dibutuhkan solusi *data forwarding*.

Clock Cycle	# 18	# 19	# 20	# 21	# 22
IF	PC = 0Ch ADDI $5, #5	PC = 0Dh AND $5, $4	PC = 0Eh SW $1, #0 ($0)	PC = 0Fh SW $2, #1 ($0)	PC = 10h SW $3, #2 ($0)
ID	ADD $4, $1	ADDI $5, #5	AND $5, $4	SW $1, #0 ($0)	SW $2, #1 ($0)
EX	MOV $5, $3	ADD $4, $1	ADDI $5, #5	AND $5, $4	SW $1, #0 ($0)
MEM	*previous Instruction (-1)*	MOV $5, $3	ADD $4, $1	ADDI $5, #5	AND $5, $4
WB	*previous Instruction (-2)*	*previous Instruction (-1)*	MOV $5, $3	ADD $4, $1	ADDI $5, #5

Gambar 2. Temuan *data hazard* #1, #2, dan #3.

Clock Cycle	# 26	# 27	# 28	# 29	# 30
IF	PC = 14h ADD $4, $1	PC = 15h NOP	*stall* *NOP*	PC = 16h LW $6, #1 ($1)	PC = 17h OR $5, $6
ID	LW $4, #2 ($0)	ADD $4, $1	*ADD $4, $1*	NOP	LW $6, #1 ($1)
EX	ADDI $2, #2	LW $4, #2 ($0)	*bubble*	ADD $4, $1	NOP
MEM	SW $3, #2 ($1)	ADDI $2, #2	LW $4, #2 ($0)	NOP	ADD $4, $1
WB	SW $3, #2 ($0)	SW $3, #2 ($2)	ADDI $2, #2	LW $4, #2 ($0)	NOP

Gambar 3. Temuan *data hazard* #4 dan #5.

https://doi.org/10.24123/saintek.v5i1.6306

Clock Cycle	# 30	# 31	# 32	# 33	# 34
IF	PC = 17h OR $5, $6	PC = 18h ADD $6, $2	*stall* *ADD $6, $2*	PC = 19h NOP	PC = 1Ah BNE $3, $1, #3
ID	LW $6, #1 ($1)	OR $5, $6	*OR $5, $6*	ADD $6, $2	NOP
EX	NOP	LW $6, #1 ($1)	*bubble*	OR $5, $6	ADD $6, $2
MEM	ADD $4, $1	NOP	LW $6, #1 ($1)	NOP	OR $5, $6
WB	NOP	ADD $4, $1	NOP	LW $6, #1 ($1)	NOP

Gambar 4. Temuan *data hazard* #6 dan #7.

Clock Cycle	# 43	# 44	# 45	# 46	# 47
IF	PC = 2Bh MOV $4, $0	PC = 2Ch MFU $4, $7	PC = 2Dh MTU $4, $7	PC = 2Eh SLL $3, #2	PC = 2Fh SLA $3, #3
ID	LWI $1, #2Bh	MOV $4, $0	MFU $4, $7	MTU $4, $7	SLL $3, #2
EX	NOP	LWI $1, #2Bh	MOV $4, $0	MFU $4, $7	MTU $4, $7
MEM	BEQ $0, $0, #7	NOP	LWI $1, #2Bh	MOV $4, $0	MFU $4, $7
WB	NOP	BEQ $0, $0, #7	NOP	LWI $1, #2Bh	MOV $4, $0

Gambar 5. Temuan *data hazard* #8 dan #9.

Clock Cycle	# 45	# 46	# 47	# 48	# 49
IF	PC = 2Dh MTU $4, $7	PC = 2Eh SLL $3, #2	PC = 2Fh SLA $3, #3	PC = 30h JLR $1	PC = 31h NOP
ID	MFU $4, $7	MTU $4, $7	SLL $3, #2	SLA $3, #3	JLR $1
EX	MOV $4, $0	MFU $4, $7	MTU $4, $7	SLL $3, #2	SLA $3, #3
MEM	LWI $1, #2Bh	MOV $4, $0	MFU $4, $7	MTU $4, $7	SLL $3, #2
WB	NOP	LWI $1, #2Bh	MOV $4, $0	MFU $4, $7	MTU $4, $7

Gambar 6. Temuan *data hazard* #10.

Pada Gambar 4, *data hazard* #6 dan #7 mempunyai kombinasi yang mirip dengan kombinasi pada *data hazard* #4 dan #5. Nilai baru untuk *register* R6 yang dibutuhkan oleh instruksi OR $5,$6 baru akan ter-*updated* saat instruksi LW $6,#1($1) telah melewati *MEM Stage*. Solusi untuk *data hazard* #6 dan #7 sama seperti solusi untuk *data hazard* #4 dan #5, yaitu *interlocking solution* yang diikuti dengan *data forwarding solution*.

Seperti yang ditunjukkan pada Gambar 5, instruksi MFU $4,$7 akan membutuhkan *register* R4 yang telah di-*upgraded* dari hasil eksekusi instruksi MOV $4,$0 saat masuk ke *EX Stage*. Namun, R4 tidak akan di-*upgraded* sampai instruksi MOV $4,$0 memasuki *WB Stage*. *Data hazard* #8 ini membutuhkan *data forwarding* sebagai solusinya. Untuk *data hazard* #9, nilai *register* R4 yang akan digunakan sebagai *operand* dari instruksi MTU $4,$7, seharusnya berasal dari hasil eksekusi instruksi MFU $4,$7 (ditunjukkan dengan garis berwarna merah), bukan berasal dari hasil eksekusi instruksi MOV $4,$0 yang ditunjukkan dengan garis putus-putus berwarna biru pada Gambar 5. Solusi *data hazard* #9 ini adalah *data forwarding* sehingga TEUS-1 dapat mem-*forward* nilai yang benar untuk instruksi MTU $4,$7 atau kombinasi instruksi lainnya yang sejenis dengan kasus ini.

Temuan *data hazard* yang terakhir (*data hazard* #10) ditunjukkan oleh Gambar 6. Instruksi SLA $3,#3 membutuhkan nilai pada *register* R3 yang telah di-*upgraded* dari hasil eksekusi instruksi SLL $3,#2 saat memasuki *EX Stage*. Namun, TEUS-1 tidak akan meng-*update*

https://doi.org/10.24123/saintek.v5i1.6306

nilai *register* R3 sampai instruksi SLL $3,#2 masuk ke *WB Stage*. *Data hazard* ini membutuhkan *data forwarding solution*.

Dari hasil analisis temuan *data hazard* #1 sampai #10, ternyata perlu untuk menambahkan *data forwarding unit* pada *EX Stage* karena data yang di-*forwarded* dibutuhkan oleh instruksi yang dieksekusi (instruksi yang memasuki *EX Stage*). Selain itu, penambahan *hazard detection unit* pada *ID Stage* untuk membuat *pipeline* dalam kondisi *stall* segera setelah ditemukan *data dependency* pada instruksi yang di-*decoded* pada *ID Stage*. Instruksi tersebut biasanya berurutan/menyertai instruksi *load* yang berada pada urutan sebelumnya dalam suatu program.

Implementasi Data Forwarding

Untuk melakukan implementasi *data forwarding* dan *hazard detection unit* pada TEUS-1, ada dua prinsip berikut ini yang seharusnya diikuti (Patterson & Hennessy, 2020a, 2020b).

1. Hasil operasi di ALU dan pembacaan data (*read data*) dari *EX/MEM pipeline register* dan *MEM/WB pipeline register* harus "dikembalikan" (*fed back*) ke *input* dari ALU.
2. Jika *data forwarding unit* mendeteksi adanya operasi ALU dan/atau operasi *load* sebelumnya yang menuliskan data (*write data*) pada suatu *register* yang menjadi *source* bagi operasi ALU berikutnya, *forwarding control logic* akan memilih hasil dari *forwarded data* sebagai *input* bagi operasi ALU daripada nilai yang dibaca dari *register file*.

Dengan mempertimbangkan kedua prinsip tersebut dan analisis *data hazard*, dibuatlah suatu daftar kondisi untuk *data forwarding control path* yang akan digunakan untuk memilih *source* bagi operasi ALU. Daftar kondisi tersebut adalah sebagai berikut.

1. Operasi sebelumnya selalu menuliskan hasilnya ke suatu *register*.
   Temuan: *data hazard* #1 – #10
   Implementasi:
   - RegWrite = '1' (dari *EX/MEM* dan *MEM/WB pipeline register*)

2. *Source register* (rs) dari operasi yang dikerjakan pada *EX Stage* harus sama dengan *destination register* (rd) dari operasi sebelumnya.
   Temuan: *data hazard* #1 – #10
   Implementasi:
   - dari *EX/MEM pipeline register*
     - EX/MEM_WriteRegister = ID/EX_rd
       atau
     - EX/MEM_WriteRegister = ID/EX_rs
   - dari *MEM/WB pipeline register*
     - MEM/WB_WriteRegister = ID/EX_rd
       atau
     - MEM/WB_WriteRegister = ID/EX_rs

3. *Destination register* dari operasi sebelumnya harus bukan *register* R0. Hal ini disebabkan karena R0 selalu bernilai nol sehingga jika R0 merupakan *destination register* maka nilai dari R0 tidak akan di-*fed back* ke *input* dari ALU walaupun kondisi #1 dan #2 terpenuhi.

   Implementasi:
   - EX/MEM_WriteRegister ≠ R0 (dari *EX/MEM pipeline register*)
   - MEM/WB_WriteRegister ≠ R0 (dari *MEM/WB pipeline register*)

4. *Source register* dari operasi sekarang pada *EX Stage* sama dengan *destination register* dari operasi sebelumnya yang dicek setelah *MEM Stage* (*MEM/WB pipeline register*) tetapi *register* tersebut tidak sama dengan *destination register* dari operasi sebelumnya yang dicek setelah *EX Stage* (*EX/MEM pipeline register*).

https://doi.org/10.24123/saintek.v5i1.6306

Temuan: data hazard #9
Implementasi:
- MEM/WB_WriteRegister = ID/EX_rd
  dengan catatan, EX/MEM_WriteRegister ≠ ID/EX_rd
- MEM/WB_WriteRegister = ID/EX_rs
  dengan catatan, EX/MEM_WriteRegister ≠ ID/EX_rs

Berdasarkan daftar kondisi dari hasil analisis *data hazard*, maka *input* untuk *data forwarding control path* adalah:
- pada *ID/EX pipeline register*:
  - ID/EX_RR1_rs, ID/EX_RR3_rd
- pada *EX/MEM pipeline register*:
  - EX/MEM_RegWrite, EX/MEM_WriteRegister
- pada *MEM/WB pipeline register*:
  - MEM/WB_RegWrite, MEM/WB_WriteRegister

Setelah merencanakan *control path* untuk *data forwarding unit*, selanjutnya, *datapath* untuk *data forwarding unit* akan diidentifikasi. *Datapath* ini akan terhubung pada *multiplexer* pada kedua *input* untuk ALU. *Datapath* sebagai *input* untuk ALU tersebut adalah:
- dari *EX/MEM pipeline register*:
  - EX/MEM_ALUResult
- dari *MEM/WB pipeline register*:
  - MEM/WB_WriteData

Dalam beberapa temuan, yaitu: *data hazard* #4 dan #6, TEUS-1 harus melakukan *stall the pipeline* untuk mengatasi *data hazard*. *Hazard detection unit* akan menguji kondisi setelah instruksi *load* dieksekusi: apakah instruksi berikutnya (berurutan) membutuhkan data yang baru saja diambil (*load/read*) dari memori. Untuk mengatasi *data hazard* akibat susunan yang berurutan dari instruksi *load* dan instruksi lainnya yang "bergantung" pada data hasil dari instruksi *load*, unit ini harus melakukan pengujian sebagai berikut.
- Apakah instruksi yang sedang dieksekusi merupakan instruksi *memory read operation (load)* dan *destination register* dari instruksi *load* tersebut sama dengan *source* atau *destination register* dari instruksi selanjutnya?
  - Implementasi:
    - ID/EX_MemRead = 1
    - ID/EX_rd = ID_RR3_rd
      atau
    - ID/EX_rd = ID_RR1_rs
- Jika kondisi tersebut terpenuhi, *hazard detection unit* harus mengirimkan *bubble* ke *EX Stage* dan menghentikan sementara proses *update* nilai *register* PC dan *IF/ID pipeline register*.
  - Implementasi:
    - mengirimkan *bubble* untuk menahan (*hold*) instruksi:
      - semua sinyal kontrol untuk EX, MEM, dan WB *Stage* = 0
      - semua *op-codes* dan *funct-codes* = 0
    - menghentikan proses *update* untuk register PC dan IF/ID *pipeline register*:
      - IF_PCWriteStall = '1'
      - IF/ID_WriteStall = '1'
- pada *cycle* selanjutnya, ketika operasi sudah bukan *memory read operation*, yang diindikasikan dengan ID/EX_MemRead = 0, *hazard detection unit* akan mengaktifkan (*wake-up*) *pipeline* dari kondisi *stall* dengan cara mengirimkan instruksi yang sebelumnya ditahan (*hold*) ke *EX Stage* dan menghapus "*stall signal*" dari *register* PC dan

https://doi.org/10.24123/saintek.v5i1.6306

IF/ID pipeline register sehingga proses eksekusi intruksi yang sebelumnya terhenti akan dilanjutkan kembali.

Dengan demikian, maka input untuk *hazard detection unit* adalah:

- dari *ID/EX pipeline register*: ID/EX_MemRead, ID/EX_rd
- dari *ID Stage*: ID_RR3_rd, ID_RR3_rs

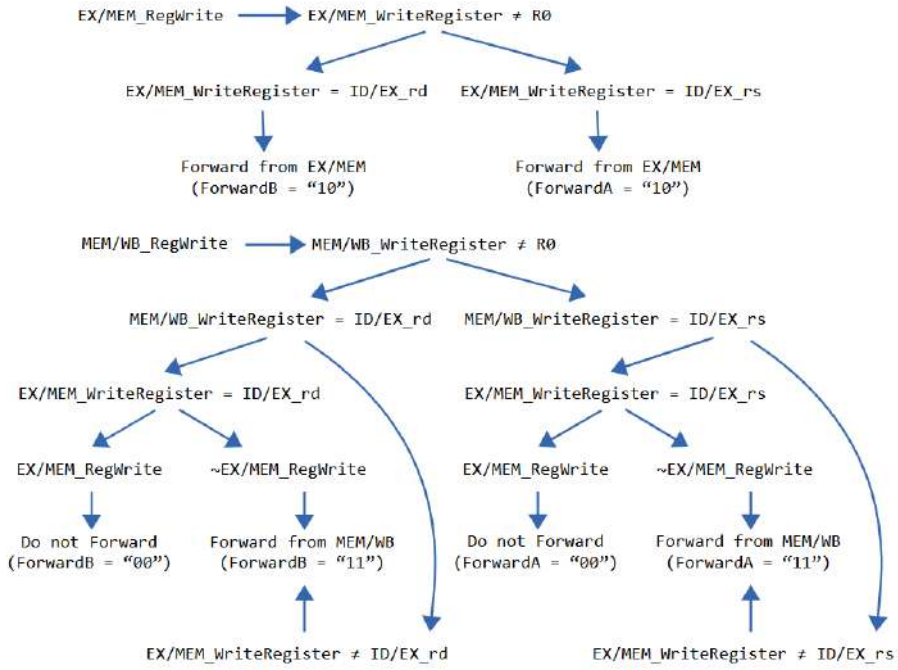
Serta, *output* dari *hazard detection unit* adalah:

- ID_CtrlStall yang digunakan untuk mengaktifkan mekanisme pengiriman *bubble* ke *EX Stage* pada *cycle* selanjutnya.
- IF/ID_WriteStall yang menonaktifkan fungsi *update* dari *IF/ID pipeline register*.
- IF_PCWriteStall yang menonaktifkan fungsi *update* dari *register* PC.

Gambar 7 dan 8 menunjukkan algoritma untuk implementasi *data forwarding unit* dan *hazard detection unit* yang telah dijelaskan sebelumnya, yang direpresentasikan dalam bentuk *graph representation*.

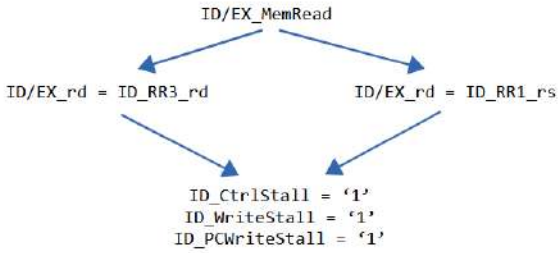
Hasil Simulasi

Dari hasil simulasi yang ditunjukkan pada Gambar 9, Gambar 10, dan Gambar 11, terlihat bahwa program telah dieksekusi dengan benar. Sebagai contoh, instruksi MOV $5,$3 yang dalam bahasa mesin direpresentasikan dengan 0x0535 (machine code = 0x0535), akan meng-*copy* nilai yang tersimpan pada *register* R3, yaitu 0xFFFF, ke *register* R5. Sebelum 0xFFFF disimpan pada R5 di *WB Stage*, ternyata data tersebut dibutuhkan oleh instruksi selanjutnya, yaitu ADDI $5,#5 (machine code = 0x6505). Dari Gambar 9 yang ditunjukkan oleh gambar panah bergaris kuning, terlihat bahwa proses *data forwarding* telah sesuai dengan rancangan, sehingga sebelum *register* R5 ter-*updated* dengan 0xFFFF, nilai 0xFFFF telah di-*forwarded* ke ALU dan selanjutnya untuk dijumlahkan dengan 0x0005 sehingga menghasilkan 0x0004 pada ALUResult. Demikian juga, nilai 0xF8A4 yang merupakan hasil eksekusi dari instruksi ADD $4,$1 (machine code = 0x0411), di-*forwarded* ke ALUData2 (*input* B dari ALU) seperti yang ditunjukkan oleh gambar panah bergaris biru muda (*cyan*) dan dapat dipastikan juga bahwa hasil dari instruksi ADDI $5,#5 (machine code = 0x6505), yaitu 0x0004, juga di-*forwarded* ke ALUData1 (ditunjukkan oleh gambar panah bergaris merah) karena dibutuhkan oleh instruksi AND $5,$4 (machine code = 0x0548) sebelum nilai 0x0004 disimpan ke *register* R5 pada eksekusi instruksi sebelumnya, yaitu MOV $5,$3.



Gambar 7. Representasi grafik dari algoritma implementasi *data forwarding unit*.

https://doi.org/10.24123/saintek.v5i1.6306



Gambar 8. Representasi grafik dari algoritma implementasi *hazard detection unit*.

Gambar 10 menunjukkan TEUS-1 melakukan *pipeline stalls* (*interlocking solution*) untuk mengatasi *data hazard* yang mempunyai pola seperti pada *data hazard* #4 dan *data hazard* #6. *Interlocking solution* terlihat dengan "diperpanjangnya keberadaan" instruksi pada *IF Stage*, *ID Stage*, dan *EX Stage* sebanyak 1 periode *clock* untuk memastikan *data dependency* terselesaikan.

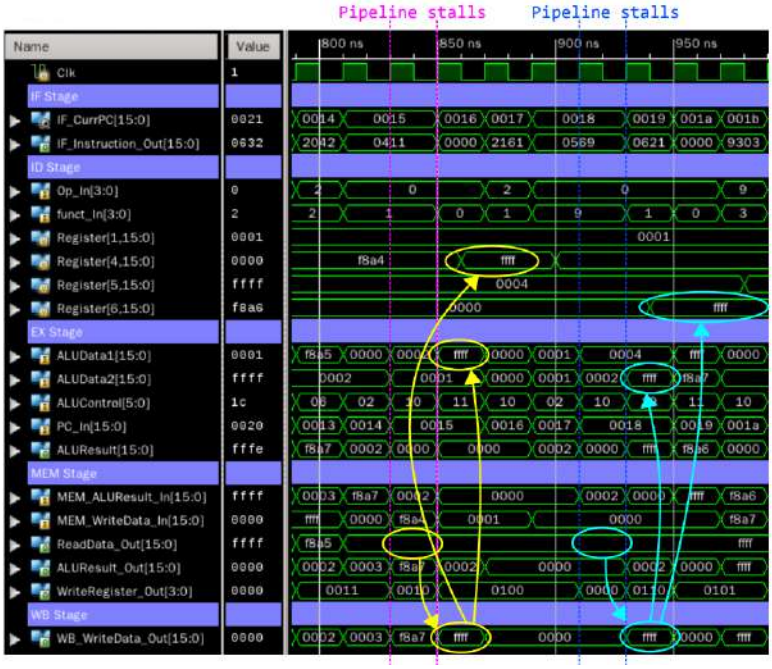
Gambar 11 menunjukkan solusi dari *data dependencies* yang saling berurutan dengan *sequence* yang cukup panjang, yaitu 4 instruksi berurutan. Hasil dari instruksi MOV $4,$0 (machine code = 0x0405), yaitu 0x0000, di-*forwarded* ke ALUData1 (*input* A dari ALU) sebagai salah satu *operand* untuk instruksi MFU $4,$7 (machine code = 0x0476). Selanjutnya, hasil dari instruksi MFU $4,$7, sebelum disimpannya nilai 0x000A ke *register* R4, nilai tersebut di-*forwarded* ke ALUData1 sebagai salah satu *operand* yang dibutuhkan untuk mengeksekusi instruksi MTU $4,$7 (machine code = 0x0477). Demikian juga, hasil eksekusi instruksi SLL $3,#2 (machine code = 0x1320) di-*forwarded* ke ALUData1 sebagai salah satu *operand* dari instruksi SLA $3,#3 (machine code = 0x1332). Dengan demikian, hasil dari eksekusi instruksi MFU, MTU, SLA, and SLL yang sekuensial dan mempunyai *data dependencies* antar instruksi tersebut adalah:

- MFU $4,$7 => R4(lo) ← R7(hi); where R4 = 0x0000, R7 = 0x0ABC
  Result: R4 = 0x000A
- MTU $4,$7 => R4(hi) ← R7(lo); where R4 = 0x000A, R7 = 0x0ABC
  Result: R4 = 0xBC0A
- SLL $3,#2 => R3 ← R3 << 0x02; where R3 = 0xFFFF
  Result: R3 = 0xFFFC
- SLA $3,#3 => R3 ← sign-ext(R3) << 0x03; where R3 = 0xFFFC
  Result: R3 = 0xFFE0

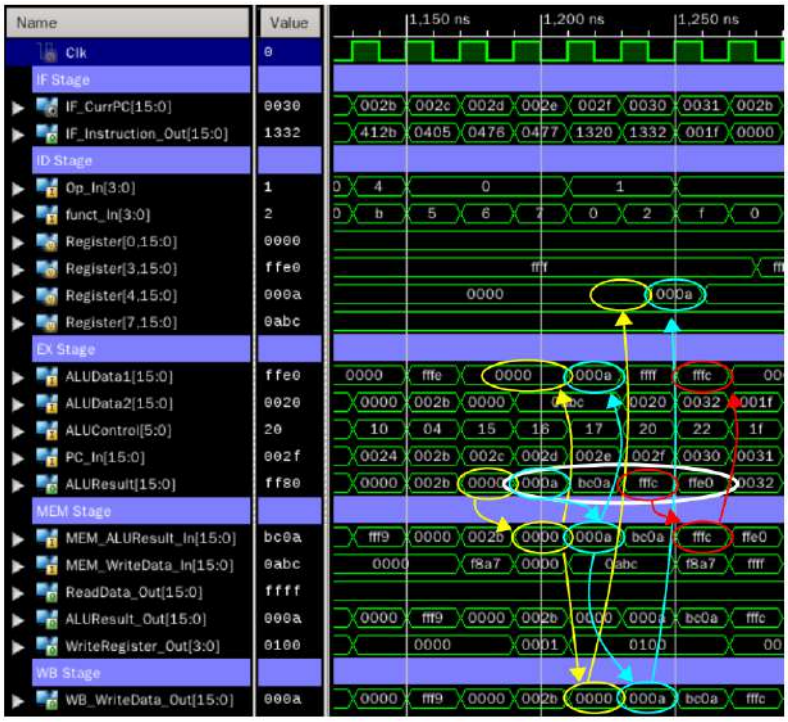
Dari hasil pengujian ini, TEUS-1 telah dapat mengeksekusi program dan mampu untuk menyelesaikan masalah *data dependencies* dengan melakukan *data forwarding* dan *interlocking solution*.

Gambar 9. Pengujian *data dependencies* (data hazard #1 - #3).

https://doi.org/10.24123/saintek.v5i1.6306



Gambar 10. Pengujian *data dependencies* (data hazard #4 - #7).



Gambar 11. Pengujian *data dependencies* (window 3: 1000ns – 1300ns).

SIMPULAN

Dari hasil pembahasan dan pengujian yang telah dilakukan, telah diimplementasikan fitur *hazard detection* dan *data forwarding* pada prosesor TEUS-1 dengan langkah implementasi yang lebih detil daripada yang telah dijabarkan pada literatur. Cara implementasi ini adalah dengan menambahkan algoritma sederhana *hazard detection* pada bagian *instruction decoding* dan *data forwarding* pada bagian *execution unit*. Penambahan ini pada intinya adalah "menyelipkan" *bubble* untuk membuat *stall* pada *pipeline* sehingga memberi kesempatan untuk melakukan *recovery data* pada *register* terkait sehingga data hasil proses instruksi sebelumnya dapat langsung digunakan oleh instruksi berikutnya.

PUSTAKA ACUAN

AMD MicroBlaze Processor: A Flexible and Efficient Soft Processor. (2023). https://www.xilinx.com/products/design-tools/microblaze.html

Barthe, L., Cargnini, L. V., Benoit, P., & Torres, L. (2011). The SecretBlaze: A Configurable and Cost-Effective Open-Source Soft-Core Processor. *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, 310–313. https://doi.org/10.1109/IPDPS.2011.154

Cheah, H. Y., Brosser, F., Fahmy, S. A., & Maskell, D. L. (2014). The iDEA DSP Block-Based Soft Processor for FPGAs. *ACM Transactions on Reconfigurable Technology and Systems*, *7*(3), 1–23. https://doi.org/10.1145/2629443

Cheah, H. Y., Fahmy, S. A., & Maskell, D. L. (2012). iDEA: A DSP block based FPGA soft processor. *2012 International Conference on Field-Programmable Technology*, 151–158. https://doi.org/10.1109/FPT.2012.6412128

Cheah, H. Y., Fahmy, S. A., Maskell, D. L., & Kulkarni, C. (2012). A lean FPGA soft processor built using a DSP block. *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 237–240. https://doi.org/10.1145/2145694.2145734

Figueroa, F. A. G. (2012). *Natalius 8 bit RISC*. https://opencores.org/projects/natalius_8bit_risc

Hermawan, H. (2012). *Leveraging FPGA Resources for Parallel System-on-Chip* [Master's Dissertation]. Nanyang Technological University.

Jean-Loup Baer. (2009). *Microprocessor Architecture: From Simple Pipelines to Chip Microprocessors*. Cambridge University Press.

Kranenburg, T., & van Leuken, R. (2010). MB-LITE: A robust, light-weight soft-core implementation of the MicroBlaze architecture. *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, 997–1000. https://doi.org/10.1109/DATE.2010.5456903

LaForest, C. E., & Steffan, J. G. (2012). OCTAVO: An FPGA-centric Processor Family. *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 219–228. https://doi.org/10.1145/2145694.2145731

Nurmi, J. (Ed.). (2007). *Processor Design: System-on-Chip Computing for ASICs and FPGAs*. Springer.

OpenCores. (n.d.). Retrieved December 1, 2023, from https://opencores.org/

Parrilla, L., García, A., Castillo, E., Álvarez-Bermejo, J. A., López-Villanueva, J. A., & Meyer-Baese, U. (2022). Dracon: An Open-Hardware Based Platform for Single-Chip Low-Cost Reconfigurable IoT Devices. *Electronics*, *11*(13), 2080. https://doi.org/10.3390/electronics11132080

Patterson, D. A., & Hennessy, J. L. (2020a). *Computer Organization and Design MIPS Edition: The Hardware/Software Interface* (Sixth Edition). Morgan Kaufmann.

Patterson, D. A., & Hennessy, J. L. (2020b). *Computer Organization and Design RISC-V Edition: The Hardware/Software Interface* (Second Edition). Morgan Kaufmann.

Schleuniger, P., McKee, S. A., & Karlsson, S. (2012). *Design Principles for Synthesizable Processor Cores* (pp. 111–122). https://doi.org/10.1007/978-3-642-28293-5_10

Shen, J. P., & Lipasti, M. H. (2013). *Modern Processor Design: Fundamentals of Superscalar Processors*. Waveland Press.

Silc, J., Robic, B., & Ungerer, T. (1999). *Processor Architecture: From Dataflow to Superscalar and Beyond*. Springer.

Tong, J. G., Anderson, I. D. L., & Khalid, M. A. S. (2006). Soft-Core Processors for Embedded Systems. *2006 International Conference on Microelectronics*, 170–173. https://doi.org/10.1109/ICM.2006.373294