BAB 3

STRATEGI PERANCANGAN

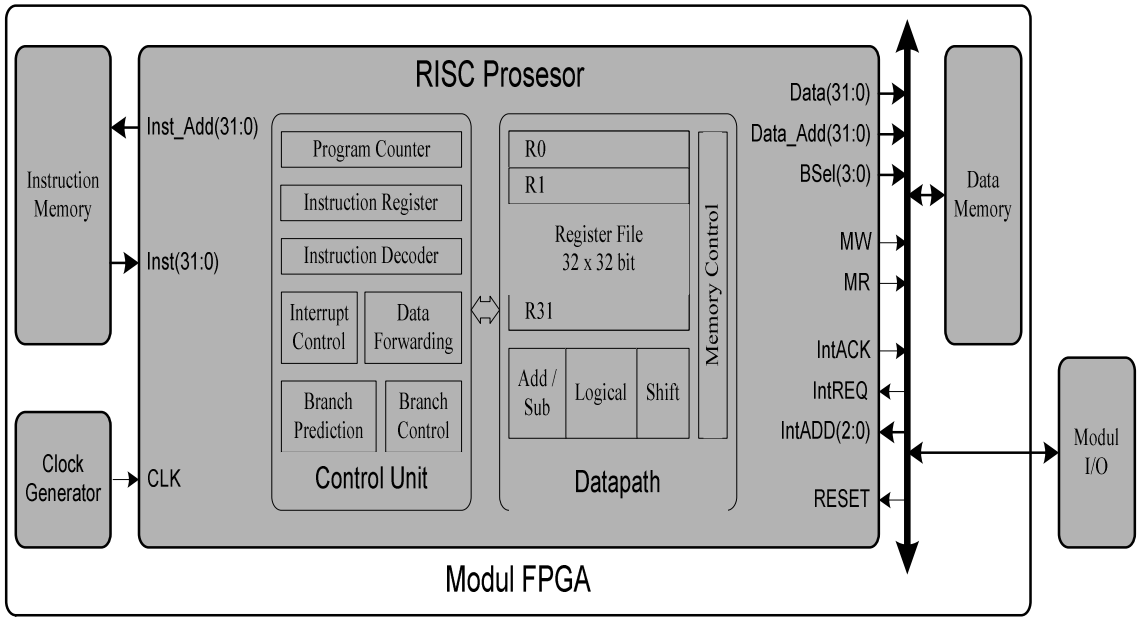
3.1 Arsitektur Prosesor

Prosesor dengan arsitektur RISC yang akan dibangun memiliki ciri-ciri sebagai berikut:

- Menggunakan arsitektur memori Harvard
- *Register File* terdiri dari 32 *register* 32 *bit*
- 32 *bit bus* data dan 32 *bit bus* instruksi
- 32 *bit bus* alamat data dan 32 *bit bus* alamat instruksi
- 13 *bit bus* kontrol *external* 15 *bit bus* kontrol *internal*
- 1Kb RAM *internal* (maksimum pengalamatan RAM 4GB)
- Maksimum pengalamatan ROM 16 GB
- Penanganan *interrupt*

Prosesor tersebut akan dibangun menggunakan modul FPGA Xilinx Spartan 2 XC2S200-PQ208, yaitu menggunakan 719 *slice* atau 30% dari keseluruhan *slice* yang tersedia dalam modul. Semua batasan perangkat keras (*hardware*) dari modul FPGA tersebut akan menjadi batasan perangkat keras dari prosesor.

Gambar 3.1 menunjukkan arsitektur dari prosesor RISC yang akan dibangun.

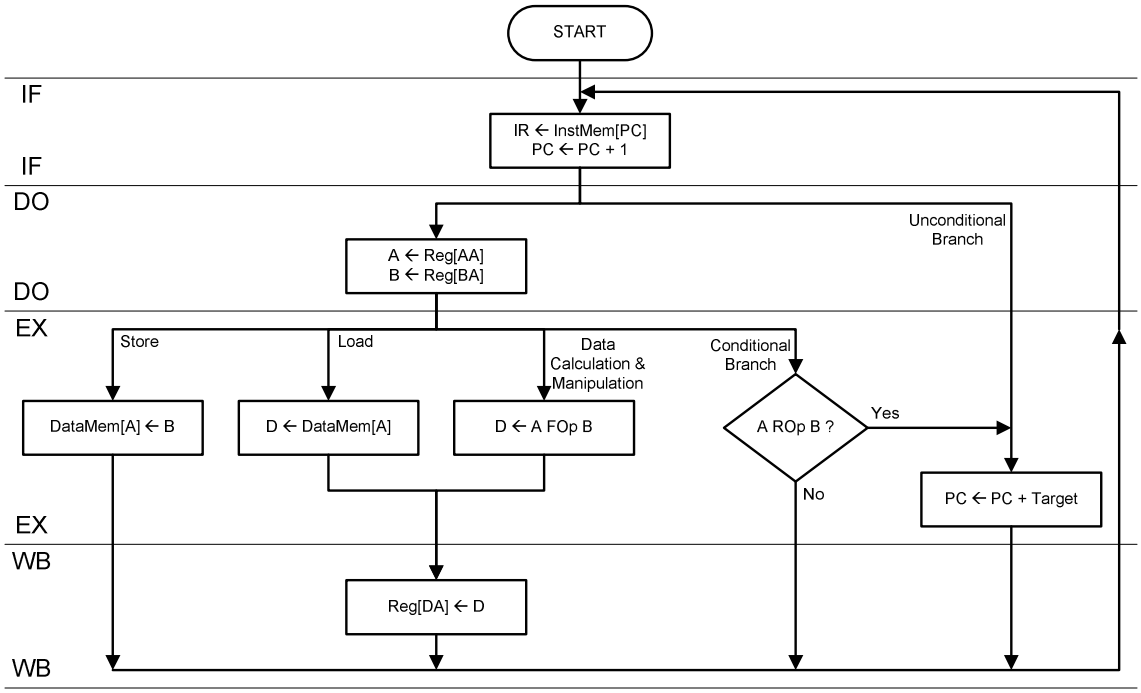


Gambar 3.1 Arsitektur Prosesor

Gambar 3.2 di bawah menunjukkan diagram alir berdasarkan pengelompokan jenis opersi. Operasi aritmatika, logika, dan pergeseran dikelompokkan menjadi *data calculation and manipulation* dengan *operand*-nya FOp (*functional operator*). Keterangan untuk Gambar 3.2 adalah sebagai berikut:

- A : Data A
- B : Data B
- D : Data D
- AA : Address A
- BA : Address B
- DA : Address D
- FOp : Arithmetic (+, -), Logic (AND, OR, XOR, NOT), dan Shift (sll, slr)

- ROp : Relational (>, <, >=, <=), Equality (==, !=)
- DataMem : Data Memory
- InstMem : Instruction Memory
- IF : Instruction Fetch
- DO : Instruction Decode and Operand Fetch
- EX : Execution
- WB : Write Back



Gambar 3.2 Diagram Alir Prosesor RISC

3.2 Rancangan Kumpulan Instruksi

Ada banyak cara yang telah diajukan dalam rangka menentukan jenis instruksi yang harus diimplementasikan pada sebuah prosesor dengan arsitektur

RISC untuk mengatasi masalah *semantic gap*. Lima langkah operasional pemilihan oleh Tanenbaum adalah contoh beberapa metodologi yang ditawarkan. (Tanenbaum, 1990)

Dalam perancangannya, pemilihan instruksi akan mengikuti jejak prosesor RISC yang telah ada, yaitu mengambil subset dari himpunan instruksi prosesor RISC yang telah ada, karena sangat dipastikan bahwa jenis instruksi dari prosesor RISC tersebut telah ditentukan mengikuti salah satu metodologi penelitian instruksi yang ada. Langkah ini diambil mengingat sasaran dari perancangan adalah sebuah dasar prosesor RISC yang dapat bekerja normal (*a working* RISC *processor*) yang merupakan lahan studi untuk sebuah perancangan prosesor RISC menggunakan teknologi FPGA. Jika teknologi dasar ini telah dikuasai, maka penambahan dan atau perubahan jenis instruksi relatif lebih mudah dilaksanakan.

Kondisi ini membuka kesempatan (proyek lain) untuk merancang sebuah prosesor RISC dengan tujuan tertentu, dimana pemilihan instruksi merupakan sentral isu dari prosesor yang akan dirancang.

Instruksi yang akan dirancang berjumlah 41 jenis instruksi dengan rincian 6 instruksi aritmatika, 8 instruksi logika, 6 instruksi pergeseran (*shift*), 8 instruksi *load/store*, 2 instruksi *set*, 2 instruksi *interrupt*, dan 9 instruksi percabangan, dimana digunakan instruksi dengan 3 *operand*.

Jenis mode pengalamatan yang digunakan ada tiga, yaitu pengalamatan *register*, *immediate* dan PC-*relative*. Untuk pengalamatan *base*, dapat dilakukan dengan menggabungkan instruksi-instruksi yang sudah ada (*pseudo instruction*).

Pipelining dibagi menjadi empat tahap yaitu *instruktion fetch* (IF), *instruction decode* dan *operand fetch* (DO), *execution* (EX), dan *write back* (WB). Untuk menghidari *pipelining hazard* digunakan beberapa metode, baik dengan *software stall*, *hadware stall*, *data forwarding*, maupun *branch prediction*.

Seperti sistem RISC umumnya, maka panjang instruksi yang digunakan adalah tetap, yaitu 32 bit, *decode* instruksi menggunakan *hardwire*, *register* berjumlah 32 buah, dan siklus instruksi yang dibutuhkan adalah 1 siklus untuk setiap instruksi. Jenis data yang digunakan adalah 32 bit alamat, 8 bit karakter, 16 dan 32 bit integer bertanda dan tidak bertanda.

Prosesor yang dirancang ini mendukung instruksi untuk operasi *stack*, penggunaan *procedure* dan penanganan *interrupt*, namun tidak mendukung instruksi perkalian dan pembagian dan juga tidak mendukung tipe data untuk bilangan berkoma (*floating point*).

Berikut adalah detail kumpulan instruksi yang dijabarkan berdasarkan aspek-aspek fundamental yang paling penting dalam merancang kumpulan instruksi (Stallings, 1997, pp9-10):

3.2.1 *Operation Repertoire*

Tabel 3.1 menunjukkan jenis-jenis instruksi yang akan dirancang dan dikelompokkan berdasarkan jenis operasinya.

Tabel 3.1 *Operation Repertoire*

Jenis Operasi	Nama Instruksi
Aritmatika	Addition
	Addition Immediate Sign
	Addition Immediate Unsign
	Subtract
	Subtract Immediate Sign
	Subtract Immediate Unsign
Logika	AND
	AND Immediate
	OR
	OR Immediate
	XOR
	XOR Immediate
	NOR
	NOR Immediate
Shift	Shift Logical Right
	Shift Logical Right Variable
	Shift Logical Left
	Shift Logical Left Variable
	Shift Arithmetic Right
	Shift Arithmetic Right Variable
Load/Store	Load Upper Immediate
	Load Address
	Load Byte
	Load Half Word
	Load Word
	Store Byte
	Store Half Word
	Store Word
Set	Set if Less Then
	Set if Less Then Immdeiate Sign
Interrupt	Disable Interrupt
	Enable Interrupt
Percabangan	Branch if Equal
	Branch if Higer
	Branch if Higer Equal
	Branch if Greater
	Branch if Greater Equal
	Jump
	Jump and Link
	Jump Register
	Jump Register and Link

3.2.2 Jenis Data (*Data Type*)

- 32 bit alamat
- 8 bit karakter ASCII
- 16 bit *integer* bertanda dan tidak bertanda
- 32 bit *integer* bertanda dan tidak bertanda

3.2.3 Format Instruksi (*Instruction Format*)

Instruksi akan dirancang dengan panjang yang tetap yaitu 32 bit untuk satu *word*. Jenis format instruksi yang digunakan:

- *Register Type*
- *Immediate Type*
- *Branch Type*
- *Jump Type*

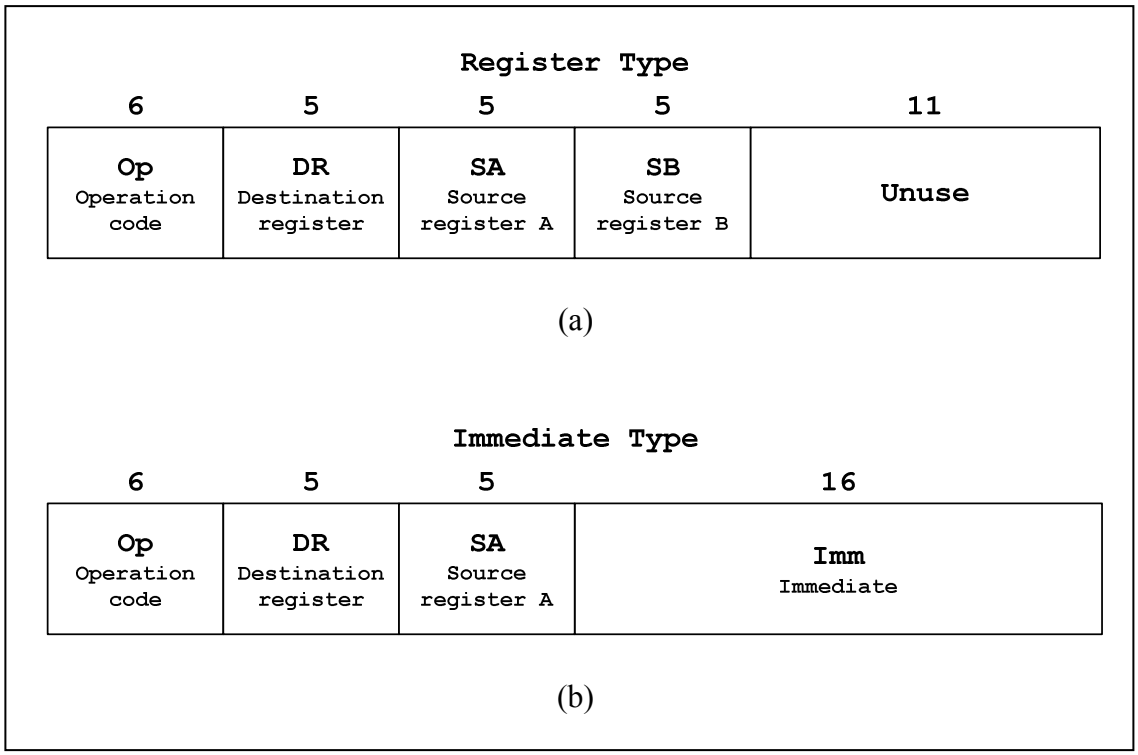
Panjang masing-masing *field* ditentukan berdasarkan faktor-faktor sebagai berikut (Stallings, 1997, pp66-67):

- Mode pengalamatan: menggunakan 3 mode pengalamatan yaitu *Register*, *Immediate*, dan PC-*Relative*.
- Jumlah instruksi: menggunakan 41 buah instruksi.
- Jumlah operand: menggunakan instruksi dengan 3 operand.
- Jumlah r*egister*: menggunakan 32 buah *register*.
- Jangkauan alamat: digunakan untuk tipe *branch* dan *jump*.

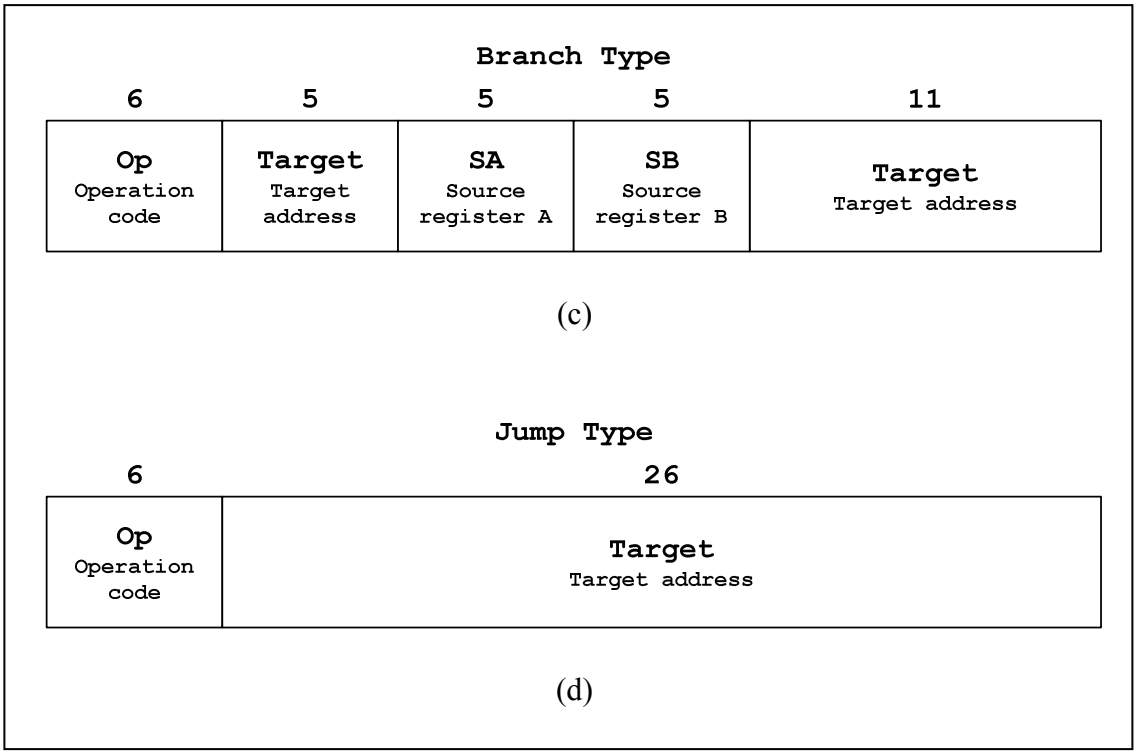
Gambar 3.3 menunjukkan pembagian *field* untuk masing-masing format instruksi dengan keterangan sebagai berikut:

- Op : 6 bit *Operation Code*
- DR : 5 bit *Destination Register*
- SA : 5 bit *Source Register* A
- SB : 5 bit *Source Register* B
- Imm : 16 bit *Immediate*
- Target : 16 bit target alamat untuk *Branch Type*

  26 bit target alamat untuk *Jump Type*



Gambar 3.3 Format Instruksi



Gambar 3.3 Format Instruksi (lanjutan)

3.2.4 *Register*

General Purpose Register (GPR) yang dirancang memiliki 32 buah 32 bit *register*. Fungsi masing-masing *register* adalah sebagai berikut:

- R0 : *Zero Register* (selalu bernilai nol)
- R29 : *Stack Pointer* (menyimpan alamat *stack* pada opersi *stack*)
- R30 : *Interrupt Return Address* (menyimpan nilai PC pada saat terjadi *interrupt*)
- R31 : *Return Address* (menyimpan nilai PC pada instruksi *jump and link* dan *jump and link register*, umumnya digunakan pada pemanggilan prosedur)

Beberapa organisasi prosesor RISC, umumnya telah menentukan fungsi dari setiap *register* berdasarkan sistem operasi yang digunakan maupun berdasarkan tujuan khusus. Dalam perncangan ini, *register* lainnya (R1 sampai R28) dapat digunakan secara bebas.

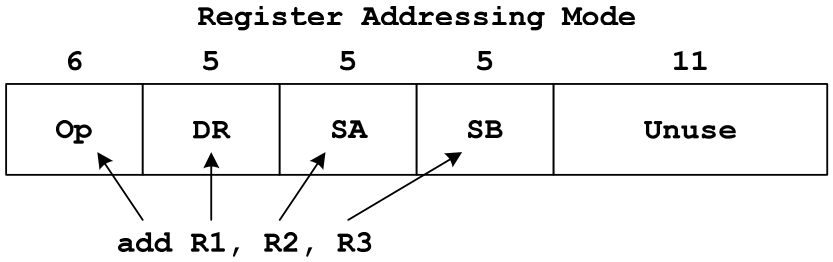
Selain GPR terdapat juga *register* dengan tujuan lain seperti *Program Counter* (untuk menyimpan alamat instruksi saat ini) dan *Instruction Register* (untuk menampung sementara instruksi sebelum di-*decode*).

3.2.5 Pengalamatan (*Addressing*)

Mode pangalamatan yang digunakan adalah:

a. *Register Addressing Mode*

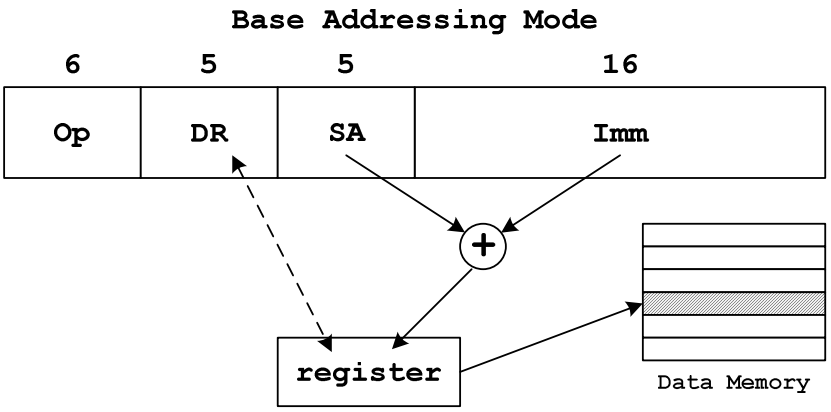
Gamabr 3.4 menunjukkan mode pengalamatan *register*.



Gambar 3.4 Mode Pengalamatan *Register*

b. *Base Addressing Mode*

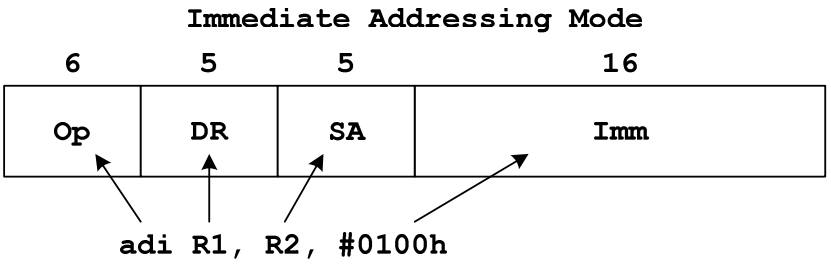
Instruksi yang mendukung mode pengalamatan basis (Gambar 3.5) tidak tersedia secara *micro instruction*, melainkan merupakan *pseudo instruction*.



Gambar 3.5 Mode Pengalamatan Basis

c. *Immediate Addressing Mode*

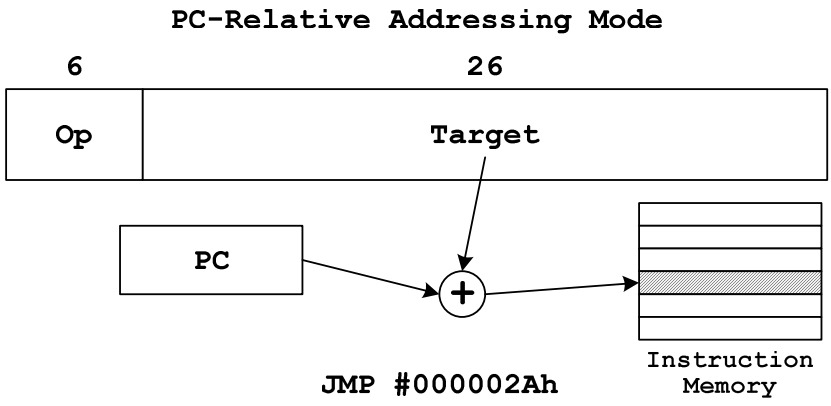
Gambar 3.6 menunjukkan mode pengalamatan *immediate*.



Gambar 3.6 Mode Pengalamatan *Immediate*

d. *PC-Relative Addressing Mode*

Instruksi percabangan (*branch* dan *jump*) dirancang menggunakan mode pengalamatan PC-*relative* (Gambar 3.7).



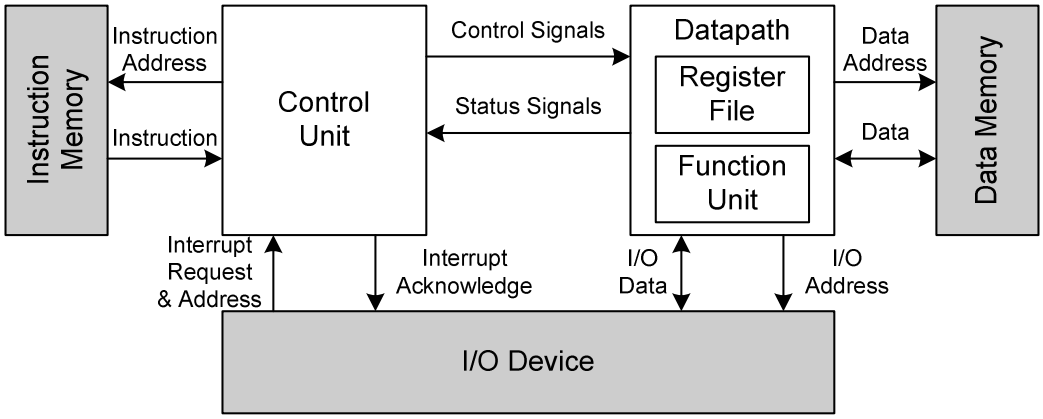
Gambar 3.7 Mode Pengalamatan PC-*Relative*

3.3 Detail Rancangan

Rancangan prosesor dibagi menjadi dua kelompok besar yaitu rancangan *control unit* dan *datapath* seperti terlihat pada Gambar 3.8. *Datapath* terdiri dari *register file* dan *function unit*, sehingga sebuah prosesor memiliki tiga komponen utama *yaitu control unit*, *register file*, dan *function unit* (*Arithmetic Logic Unit*/ALU).

Control unit berfungsi untuk mengatur jalannya operasi pada prosesor dimana memperoleh masukkan berupa kondisi dari *datapath* melalui sinyal status untuk memutuskan instruksi selanjutnya yang akan diambil. *Control unit* akan mengeluarkan alamat instrksi ke memori instruksi untuk mengambil instruksi yang akan dikerjakan sebagai masukkannya. Masukkan lain dari *control unit* berupa permintaan *interrupt* dan alamat *interrupt* dari peralatan I/O (*Input*/*Output*). Alamat *interrput* akan menuju tabel *interrupt* dan menentukan jenis *interrupt* yang akan dieksekusi. Kemudian instruksi akan dikodekan dan dikirim melalui sinyal kontrol ke *datapath* untuk memberitahukan apa yang

harus dikerjakan oleh *datapath*. *Datapath* memiliki keluaran berupa alamat atau data ke memori data atau peralatan I/O dan memiliki masukkan berupa data dari memori data maupun peralatan I/O.



Gambar 3.8 Interaksi antara *Datapath* dan *Control Unit*

3.3.1 *Register File*

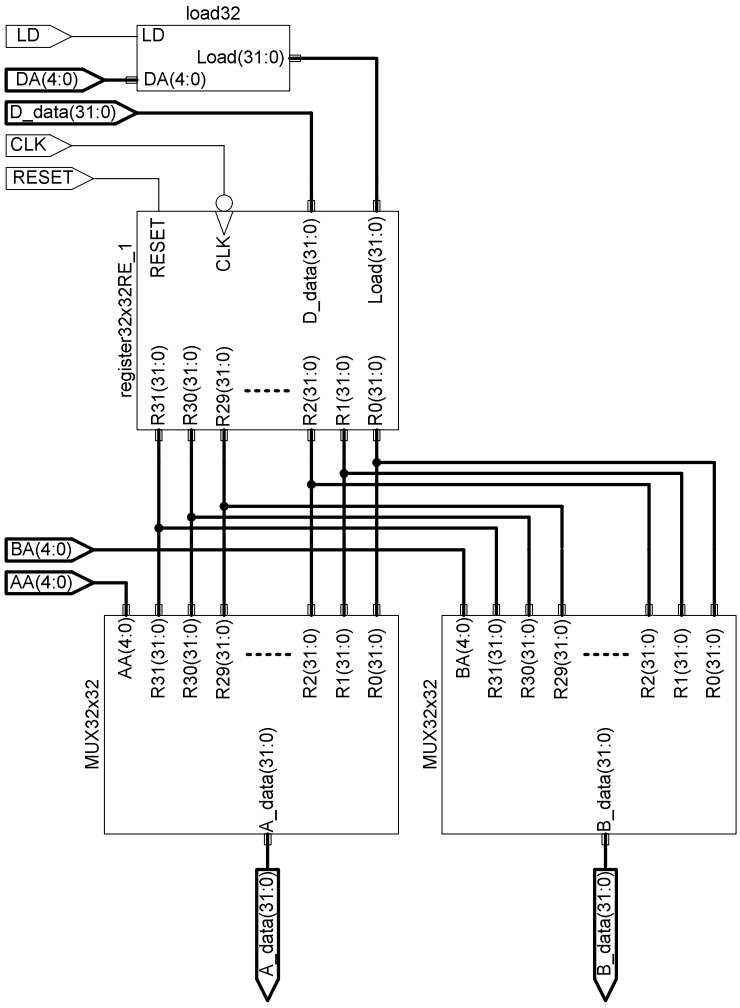
Register file merupakan kumpulan *register* yang digunakan sebagai tempat penyimpanan dan pengambilan data dalam melakukan operasi sebuah instruksi. *Register file* yang dirancang memiliki 32 buah 32-bit *register*. Dimana *register* nol (R0) akan selalu bernilai nol pada saat pengambilan data dan jika dilakukan penulisan data ke *register* nol, maka data tersebut akan dibuang.

Register dapat dibangun menggunakan *flip-flop* maupun menggunakan *Dual Port Random Access Memory* (DP RAM). *Flip-flop* yang digunakan adalah *storage element* yang terdapat pada setiap *logic cell* dimana sebuah *storage element* hanya dapat menampung 1 bit data. Sebuah

logic cell hanya memiliki satu *storege element*, sehingga dalam satu CLB hanya terdapat empat buah *storage element* yang hanya dapat menampung 4 bit data.

FPGA Spartan II produksi Xilinx mengunakan *dual port* RAM yang sinkron untuk diimplementasikan pada setiap LUT-nya. Setiap DP RAM dapat dialamati oleh dua buah alamat, yang pertama adalah alamat untuk menulis dan yang kedua adalah alamat untuk membaca. Karena akan digunakan instruksi dengan tiga operand yang pada satu saat dibutuhkan dua alamat untuk membaca dengan target yang mungkin berbeda, maka dibutuhkan dua kali jumlah DP RAM yang duplikasi (menjadi 64 *register* 32 bit). Pada saat menulis, alamat dan data akan ditulis ke kedua DP RAM, sedangkan pada saat membaca, alamat ditujukan dan data diambil dari masing-masing DP RAM dengan target dan hasil data yang dapat berbeda. Sebuah 4 bit LUT memiliki 16 bit RAM, maka sebuah CLB yang memiliki 4 buah 4 bit LUT dapat menampung hingga 64 bit data.

Untuk perancangan menggunakan *flip-flop* dibutuhkan komponen *load* untuk menunjuk *register* mana yang akan ditulis dari ke 32 *register*. Komponen *load* mendapatkan *input* berupa sinyal (LD) yang menandakan apakah terjadi penulisan ke *register*. Pada saat pembacaan dibutuhkan dua buah *multipexer* 32 *input* 32 bit untuk masing-masing data *operand* yang ditunjuk oleh alamat *operand* tersebut. Komponen *register* menggunakan *reset* asinkron. Skematik untuk *register file* menggunakan *flip-flop* dapat dilihat pada Gambar 3.9.

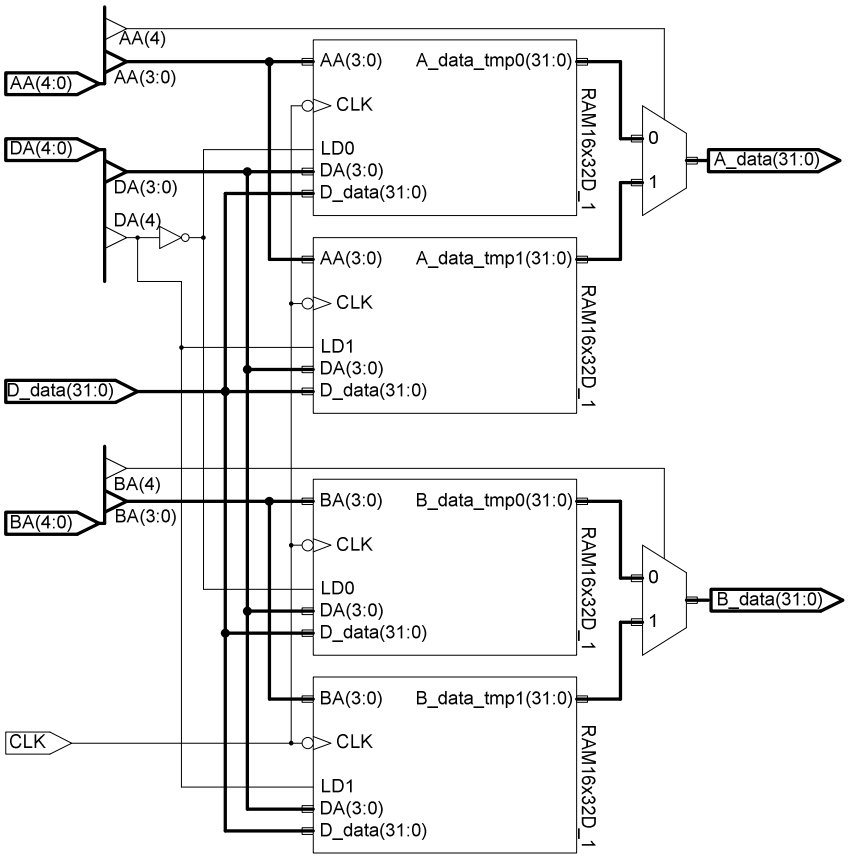


Gambar 3.9 *Register File* dengan *Flip-flop*

Tidak terdapat sinyal *reset* pada perancangan *register file* menggunakan DP RAM. Perancangan dilakukan menggunakan *library* RAM16X1D_1 (*16-Deep by 1-Wide Static Dual Port Synchronous RAM with Negative-Edge Clock*). Skematik untuk *register file* menggunakan DP RAM dapat dilihat pada Gambar 3.10.

Perancangan *register file* menggunakan *negatif edge clock* karena *register file* menggunakan metoda *read after write*, dimana penulisan dan

pembacaan ke dan dari *register* dilakukan dalam satu *clock*. *Register* akan ditulis pada *low edge* dan dibaca pada *high edge*, maka persiapan penulisan *register* adalah pada setegah *clock* pertama dan persiapan pembacaannya pada setengah *clock* kedua.



Gambar 3.10 *Register File* dengan *Dual Port* RAM

3.3.2 *Function Unit*

Function unit melakukan operasi aritmatika (*addition* dan *subtraction*), logika (AND, OR, XOR, NOR), pergeseran (aritmatika dan logika), serta *load upper immediate* (LUI).

3.3.2.1 *Addition*

Jenis-jenis *adder* yang dirancang untuk dijadikan perbandingan adalah sebagai berikut:

- *Ripple Carry Adder*

  Merupakan penjumlahan yang paling sederhana dimana digunakaan *full adder* untuk mengambil *carry* dari penjumlahan sebelumnya. Penjumlahan bit ke 32 harus menunggu ke 31 penjumlahan bit sebelumnya selesai, sehingga TPD (*Time Propagation Delay*) yang dihasilkan akan bertambah secara linier sesuai dengan panjangnya bit yang akan dijumlahkan.

  Dalam perancangannya akan digunakan metode struktural dimana 32 *full adder* dirancang terlebih dahulu kemudian masing-masing *pin* dihubungkan sehingga membentuk *ripple carry adder*.

- *Carry Lookahead Adder*

  Merupakan perbaikan dari *ripple carry adder* dengan menabah fungsi untuk memajukan *carry*, sehingga penjumlahan selanjutnya tidak perlu menunggu penjumlahan sebelumnya selesai.

  Perancangan dilakukan dengan memisahkan komponen penjumlahan dengan komponen yang menghasilkan *carry*. Komponen yang hanya melakukan penjumlahan disebut *partial full adder* dan komponen yang memajukan nilai *carry* disebut *carry lookahead*.

Untuk memajukan *carry* sejauh n bit dibutuhkan gerbang AND dan OR yang memiliki n *input*. Untuk penjumlahan 32 bit, maka penjumlahan bit ke 32 membutuhkan gerbang AND dan OR dengan 32 input untuk memajukan *carry* dari bit ke 0. Hal ini mengakibatkan bengkaknya jumlah CLB yang dibutuhkan dan kecepatan yang dihasilkan tidak meningkat jauh karena jumlah input maksimum untuk LUT hanya 4.

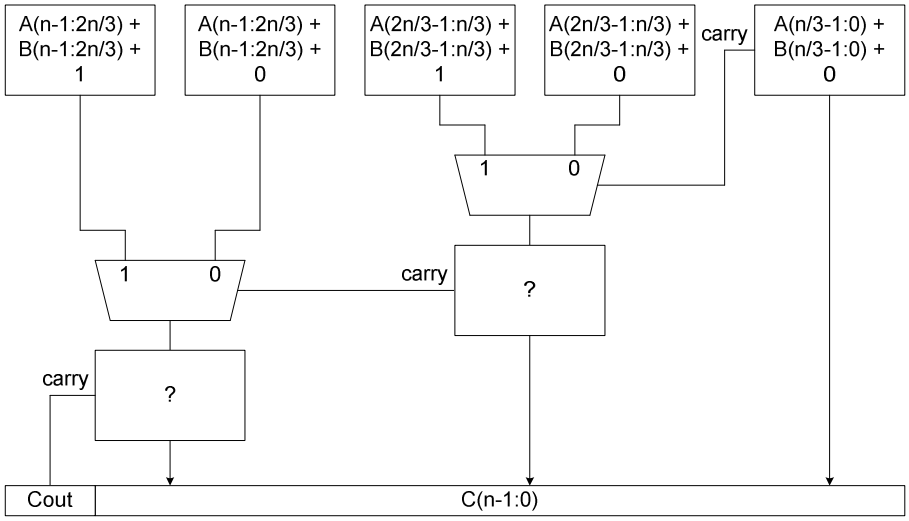
- *Carry Selector Adder*

  Penjumlahan dibagi menjadi x bagian. Masing-masing bagian melakukan penjumlahan dengan dua kemungkinan, yaitu dengan *carry in* sama dengan 1 dan 0. Jika ternyata bagian sebelumnya menghasilkan *carry out* sama dengan 1, maka hasil penjumlahan dengan *carry in* sama dengan 1 yang akan dilewatkan melalui *multiplexer* dengan 2 *input* sebagai hasil yang diperoleh, begitu juga sebaliknya. Oleh karena itu dibutuhkan x-1 *multiplexer* dengan 1 bit selektor yang merupakan *carry out* hasil penjumlahan sebelumnya.

  Maka TPD total yang dihasilkan adalah:

  TPD(total)=[TPD(penjumlahan n/x bit)]+[(x-1)xTPD(MUX)]

  Pada perancangan akan digunakan x sama dengan 2. Berikut adalah contoh dimana x sama dengan 3 pada Gambar 3.11.



Gambar 3.11 Contoh *Carry Selector Adder* dengan x = 3

- *Adder* Menggunakan *Library*

  Xilinx telah menyediakan kemampuan menggunakan *library* pada *software*-nya dengan tujuan untuk mempermudah dalam melakukan perancangan. *Library* untuk *adder* disediakan oleh IEEE yang tersimpan pada *file* STD_LOGIC_UNSIGNED.VHD.

3.3.2.2 *Adder* and *Subtractor*

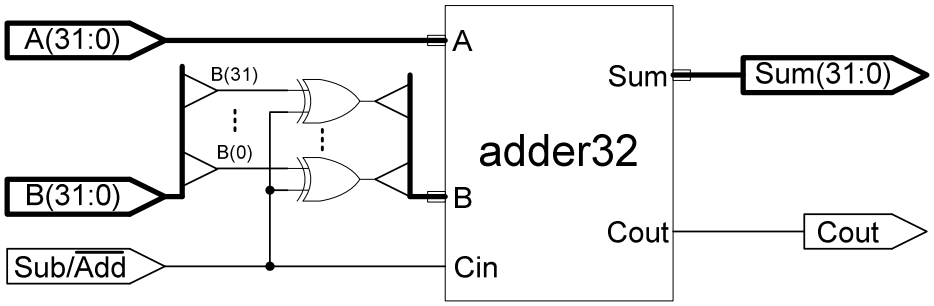
Subtractor dapat disatukan dengan rangkaian *adder* dengan menggunakan motoda *two's complement* pada nilai yang akan mengurangkan agar nilai tersebut menjadi negatif, selanjutnya nilai tersebut akan ditambahkan dengan nilai yang akan dikurangkan. Berikut ini adalah persamaannya:

$$RD \leftarrow RA - RB$$

$$RD \leftarrow RA + (-RB)$$

$$RD \leftarrow RA + (\overline{RB} + 1)$$

Karena itu dibutuhkan *adder* yang memiliki *carry in* dimana nilainya sama dengan satu pada operasi pengurangan dan nol pada operasi penjumlahan. Dibutuhkan juga pembalik inputan pengurang yang dapat dilakukan dengan penambahan gerbang XOR dua masukan dimana masukan pertama adalah nilai pengurang dan masukan kedua adalah bernilai satu untuk operasi pengurangan dan nol untuk operasi penjumlahan. Gambar 3.12 menunjukkan skematik rangkaian *adder* dan *subtractor*.



Gambar 3.12 Skematik *Adder* dan *Subtractor*

Ada kemungkinan saat data harus dilewatkan tanpa terjadi penambahan dan pengurangan seperti pada operasi yang menyimpan nilai PC (*Progaram Counter*) ke *register*. Hal ini dapat dilakukan dengan membuat nol nilai penambah atau pengurang.

Tabel 3.2 berikut merupakan tabel operasi penambahan, pengurangan, dan pelewatan data.

Tabel 3.2 Operasi Penambahan, Pengurangan, dan Pelewatan Data

Pass	Sub/$\overline{\text{Add}}$	Operation
0	0	Adder
0	1	Subtractor
1	X	Pass

Cara lain adalah dimana perancangan dilakukan sepenuhnya dengan *library* untuk penambahan dan *library* untuk pengurangan.

3.3.2.3 Logika

Empat instruksi dasar logika yaitu AND, OR, XOR, dan NOT. Instruksi logika NOT dapat dibentuk dari logika NAND maupun NOR. Berikut adalah persamaannya:

$$\text{NOT A} \Leftrightarrow \text{A NAND 1} \Leftrightarrow \text{A NOR 0}$$

Pada perancangan akan digunakan NOR sebagai pengganti NOT karena NOR dapat berfungsi sebagai NOR maupun NOT. Tabel 3.3 merupakan tabel untuk operasi logika.

Tabel 3.3 Operasi Logika

S	Operation
00	AND
01	OR
10	XOR
11	NOR

3.3.2.4 *Arithmetic Logic Unit* (ALU)

ALU merupakan gabungan dari komponen aritmatika (*adder* dan *subtractor*) dan logika (AND, OR, XOR, NOR). Pengambungan kedua komponen ini hanya ditujukan untuk kesederhanaan perncangan, dimana komponen ALU tidak menyatukan keluaran untuk operasi aritmatika dan logika. Hal ini ditujukan untuk memperkecil TPD yang dihasilkan oleh *function unit* dengan cara menggunakan satu *multiplexer* empat input untuk operasi aritmatika, logika, *shifter*, dan LUI.

Selain aritmatika dan logika, ALU juga memiliki keluaran berupa empat *flag* yaitu *carry* (C), negatif (N), *overflow* (V), *zero* dan XNV yang dihasilkan dari operasi aritmatika. XNV merupakan XOR antara *flag* negatif dengan *overflow* yang menunjukkan apakah *output* yang dihasilkan oleh operasi aritmatika adalah negatif tetapi tidak *overflow*. Nilai yang dihasilkan oleh XNV dapat disimpan pada *register* yang ditunjuk dan digunakan pada operasi *set*.

Flag menunjukkan kondisi keluaran pada operasi aritmatika. Kaluaran merupakan hasil yang salah jika pada penjumlahan bilangan tidak bertanda *carry* yang dihasilkan bernilai 1, pada pengurangan bilangan tidak bertanda jika *carry* yang dihasilkan bernilai 0, penjumlahan dan pengurangan bertanda jika *overflow* bernilai 1.

Tabel 3.4 menunjukkan sinyal yang digunakan pada saat perancangan ALU untuk menentukan jenis operasi.

Tabel 3.4 Operasi ALU

Function	Output	FS	Operation
Arithmetic	AS	00	Adder
	AS	01	Subtractor
	AS	10	Pass
	AS	11	Pass(X)
Logic	LU	00	AND
	LU	01	OR
	LU	10	XOR
	LU	11	NOR

3.3.2.5 *Shifter*

Instruksi pergeseran yang akan dirancang yaitu *logical shift left*, *logical shift right*, dan *arithmetic shift right*. *Arithmetic shift right* digunakan pada algoritma *booth* untuk membentuk instruksi perkalian. Banyaknya pergeseran yang dilakukan diambil dari lima bit LSB dari nilai *immediate* maupun *register*. Penamaan untuk banyaknya pergeseran yang diambil dari nilai *register* disebut *shift variable*.

Perancangan dilakukan menggunakan dua jenis *shift*, yaitu *shifter* biasa dan *barrel shifter*. Pada perancangan menggunakan *barrel shifter* akan dibandingkan jika menggunakan satu dan dua bit selektor pada *multiplexer*-nya.

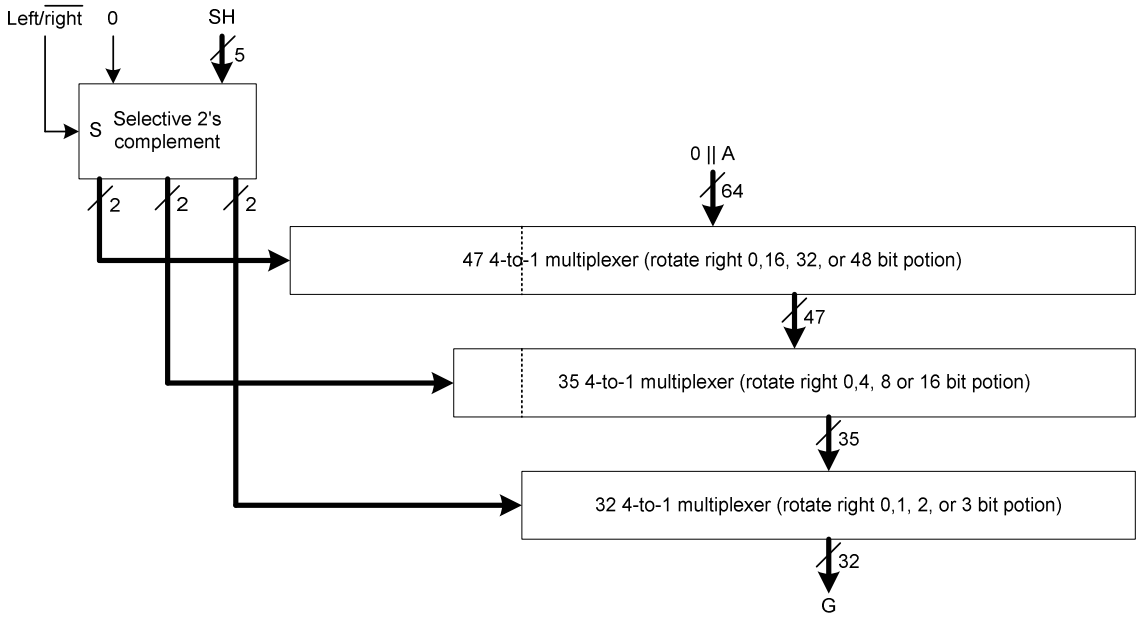
Karena pada dasarnya *barrel shifter* hanya melakukan rotasi, maka untuk *shift* sebanyak n bit dibutuhkan n bit tambahan yang berisi nol untuk mengantisipasi jika terjadi shift sebanyak n bit. Cara kerjanya menyerupai *binary tree*. Jika digunakan 2 bit selektor, maka total 2n bit yang akan digeser dibagi menjadi empat bagian kemudian *multiplexer* empat input dengan 2 bit selektor akan menentukan pergeseran bit-bit

tersebut. Dibutuhkan juga komponen *two's complement* untuk menentukan arah pergeseran (ke kiri atau ke kanan).

Contoh dimana n sama dengan 32 bit dan 2 bit selektor, maka dengan tambahan 32 bit nol totalnya menjadi 64 bit. Untuk total 64 bit, selektor *multiplexer* yang dibutuhkan adalah 6 bit. Tahap pertama adalah membagi 64 menjadi 4 bagian, yaitu 0, 16, 32, atau 48 dengan panjang yang sama yaitu 16 bit. Tahap kedua adalah membagi 16 bit menjadi 4 bagian 0, 4, 8, atau 12 dengan panjang sama yaitu 4 bit. Tahap ketiga membagi 4 bit menjadi 4 bagian, yaitu 0, 1, 2, dan 3 dengan panjang yang sama yaitu 1 bit. Jumlah *multiplexer* di tahap ketiga adalah 32 buah karena output yang diinginkan adalah 32 bit. Jumlah *multiplexer* tahap kedua adalah 32 ditambah 3 (posisi terjauh tahap ketiga) sama dengan 35 *multiplexer*. Jumlah *multipexer* tahap pertama adalah 35 ditambah 12 (posisi terjauh tahap kedua) sama dengan 47 *multiplexer*.

Gambar 3.13 merupakan contoh *barrel shifter* dengan 32 bit *input* dan 2 bit selektor yang juga digunakan dalam perancangan.

Untuk instruksi *logical shift* maka n bit tambahan diisi dengan nol sedangkan untuk *arithmetic shift* n bit tambahan diisi dengan nilai yang sama dengan bit MSB untuk pergeseran ke kanan dan LSB untuk pergeseran ke kiri.



Gambar 3.13 *Barrel Shifter* dengan 32 bit *input* dan 2 bit selektor

Tabel 3.5 menunjukkan sinyal yang digunakan pada saat perancangan *shifter* untuk menentukan jenis operasi.

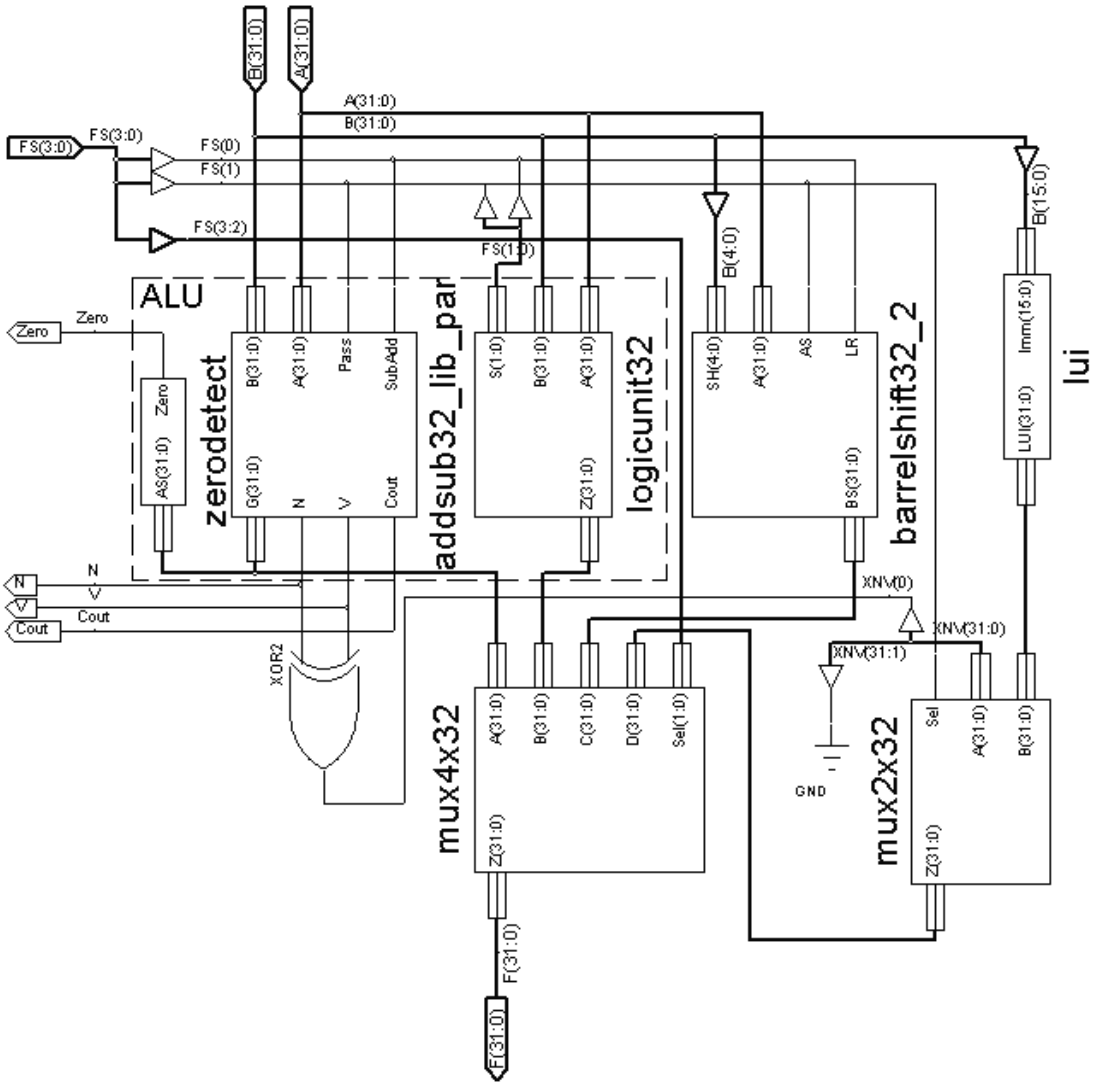
Tabel 3.5 Operasi Pergeseran

AS	LR	Operation
0	0	Logical Shift Right
0	1	Logical Shift Left
1	0	Arithmetic Shift Right
1	1	Arithmetic Shift Left (X)

3.3.2.6 *Load Upper Immediae* (LUI)

LUI digunakan untuk mempercepat load data dari *immediate* (16 bit LSB) ke sebuah *register* pada posisi 16 bit MSB. Cara lain adalah dengan menggunakan *logical shift left* sebanyak 16 bit.

Setelah semua komponen *function unit* di rancang dan didapatkan TPD dan jumlah *slice* masing-masing komponen dengan bantuan *software*, maka perancangan *function unit* sudah dapat dilakukan. Gambar 3.14 merupakan skematik dari *function unit*.



Gambar 3.14 Skematik *Function Unit*

Tabel 3.6 menunjukkan sinyal yang digunakan dalam perancangan *function unit* untuk menentukan jenis operasi.

Tabel 3.6 Operasi pada *Function Unit*

Function	FS	Operation
Arithmetic	00 00	Adder
	00 01	Subtractor
	00 10	Pass
	00 11	(X)
Logic	01 00	AND
	01 01	OR
	01 10	XOR
	01 11	NOR
Shifter	10 00	Logical Shift Right
	10 01	Logical Shift Left
	10 10	Arithmetic Shift Right
	10 11	(X)
SET	11 0X	XNV
Load	11 1X	Load Upper Immediate

3.3.3 *Datapath*

Instruksi yang dirancang menggunakan tiga *operand*, satu sebagai target dan dua sebagai sumber. Kedua *operand* sumber akan melewati *bus* A dan *bus* B yang terhubung dengan masukkan dari *function unit* untuk dieksekusi. Hasil dari eksekusi akan disimpan ke dalam *register* yang ditunjuk oleh *operand* target melalui *bus* D.

Karena data dari *operand* sumber yang melalui *bus* B dapat berasal dari nilai *immediate* maupun *register* pada *register file*, maka dibutuhkan sebuah *multiplexer* dengan dua *input* untuk memilih kedua kemungkinan ini. *Multiplexer* untuk memilih masukkan pada *bus* B ini disebut MUX B.

Begitu juga dengan jalur pada *bus* A. Selain berasal dari nilai *register* pada *register file*, *bus* A juga digunakan untuk melewatkan nilai PC (*Program Counter*) yang ingin disimpan di *register* (digunakan untuk instruksi *load address* (LA), *jump and link* (JL), dan *jump register and link* (JRL)). *Multiplexer* dua *input* untuk memilih kedua kemungkinan ini disebut MUX A.

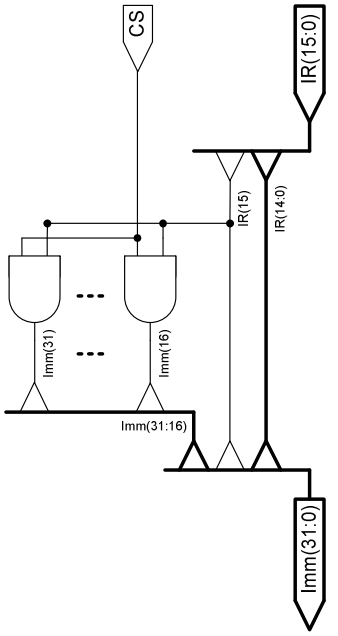
Salah satu ciri dari arsitektur RISC adalah tidak terdapat operasi yang menggabungkan operasi *load*/*store* dengan operasi aritmatika. Oleh karena itu operasi pengambilan atau penyimpanan data dari dan ke memori data dapat diparalel dengan eksekusi pada *function unit*.

Pada saat melakukan operasi *load*/*store* dimana tidak terjadi operasi pada *function unit* maka *bus* A dapat digunakan sebagai jalur alamat. Untuk instruksi *store*, *bus* B dapat digunakan sebagai jalur data ke memori data. Sedangkan untuk instruksi *load*, data yang diperoleh dari memori data langsung disimpan di *register* melalui *bus* D. Karena hasil eksekusi pada *function unit* dan data dari memori data disimpan di *register* melalui *bus* D maka dibutuhkan *multiplexer* dua *input* pada jalur ini yang disebut MUX D.

Pada instruksi *jump register* dan *jump register and link*, nilai PC yang baru diambil dari *register*. Nilai ini akan dilewatkan melalui *bus* B karena *bus* A digunakan pada instruksi *jump and link* untuk menyimpan nilai PC saat ini ke *register*.

Terdapat instruksi-instruksi aritmatika seperti penjumlahan dan pengurangan *immediate* menggunakan nilai bertanda dan tidak bertanda. Karena masukkan dari nilai *immediate* hanya 16 bit (LSB) maka untuk

mengkonversi ke 32 bit, ke 16 bit MSB harus diisi dengan nilai yang benar agar tidak merubah nilai *immediate* tersebut. Komponen untuk menangani hal ini disebut *constant unit* (Gambar 3.15). *Constant unit* memiliki masukkan yang menandakan apakah bilangan *immediate* yang akan dikonversi adalah bilangan bertanda atau tidak. Jika ternyata bukan bilangan bertanda maka ke 16 bit MSB diisi dengan nol. Jika bilangan bertanda maka ke 16 bit MSB diisi sama dengan bit ke 15 (MSB dari *immediate*).



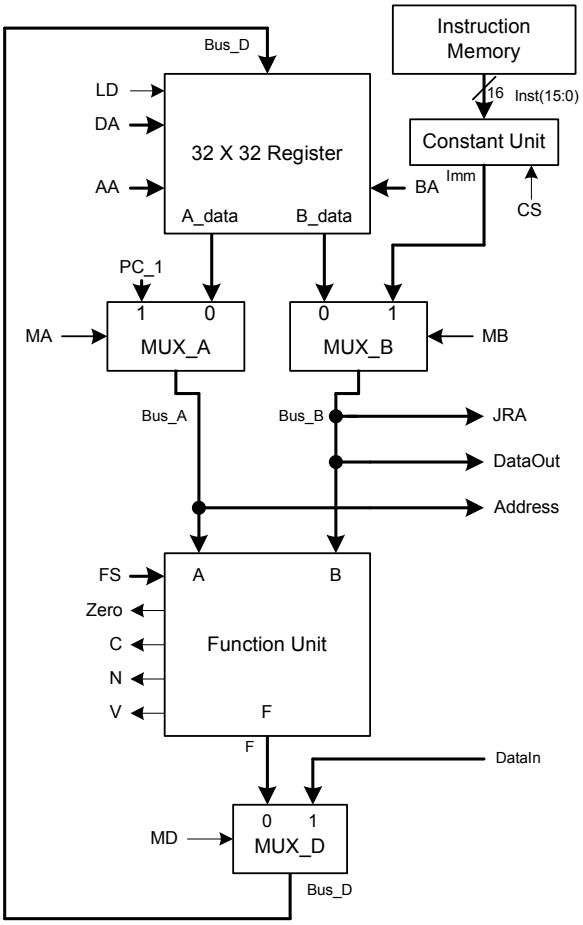
Gambar 3.15 Skematik *Constant Unit*

Gambar 3.15 dan tabel 3.7 merupakan skematik dan sinyal yang memilih operasi dari konstan unit.

Tabel 3.7 *Constant Unit*

CS	Operation
0	X"0000" || IR(15:0)
1	IR(15) || IR(15:0)

Gambar 3.16 merupakan skematik dari *data path*. *Data path* tidak dikelompokan menjadi komponen tersendiri melainkan disatukan dengan komponen *control unit*.



Gambar 3.16 Skematik *Datapath*

Untuk kederhanaan perancangan, *register file*, MUX A, MUX B, dan *constant unit* disatukan menjadi komponen *register file* gabungan.

3.3.4 *Memory Control*

Untuk menghemat memori data maka penyimpanan dapat dilakukan dalam beberapa variasi bit yaitu 8, 16 dan 32 bit. Instruksi untuk menyimpan data sebesar 8 bit ke memori data disebut *store byte* (SB), untuk 16 bit disebut *store half* (SH), dan 32 bit disebut *store word* (SW). Begitu juga pada saat pengambilan data dari memori data, *load byte* (LB) untuk mengambil data sebesar 8 bit, *load half* (LH) sebesar 16 bit, dan *load word* (LW) sebesar 32 bit.

Karena data terkecil adalah 8 bit (1 byte) maka setiap alamat menunjuk pada data yang besarnya 8 bit. Pada perancangan ini digunakan *litle endian* dalam menyimpan data pada memori data. Untuk mengatur penyimpanan data pada memori data digunakan komponen *memory control out* (MCO) sedangkan untuk pengambilan data digunakan komponen *memory control in* (MCI).

Tabel 3.8 merupakan sinyal yang digunakan untuk memilih operasi *load*/*store*.

Tabel 3.8 Sinyal untuk Opersi *Load*/*Store*

LS(1:0)	Load Opertion	Store Opertion
00	Load Byte (LB)	Store Byte (SB)
01	Load Half Word (LH)	Store Half Word (SH)
10	Load Word (LW)	Store Word (SW)

Alamat yang digunakan untuk menunjuk setiap *word* data berjumlah 30 bit MSB sedangkan 2 bit LSB digabungkan bersama dengan sinyal LS untuk menentukan bagian mana dari *word* yang akan disimpan dan diambil.

Tabel 3.9 merupakan pemetaan *register* ke memori data dimana E0 sampai dengan E3 adalah satu *word* yang panjangnya masing-masing adalah 8 bit. Angka satu menunjukkan lokasi memori yang dapat ditulis pada masing-masing operasi dan alamat. Selain kombinasi tersebut, penulisan tidak diijinkan (dapat menyebabkan kerusakan data), contohnya operasi LH pada alamat 01. Tabel ini menunjukkan operasi yang dilakukan oleh *decoder* pada Gambar 3.17.

Tabel 3.9 Dekoder untuk *Enable* pada MCO

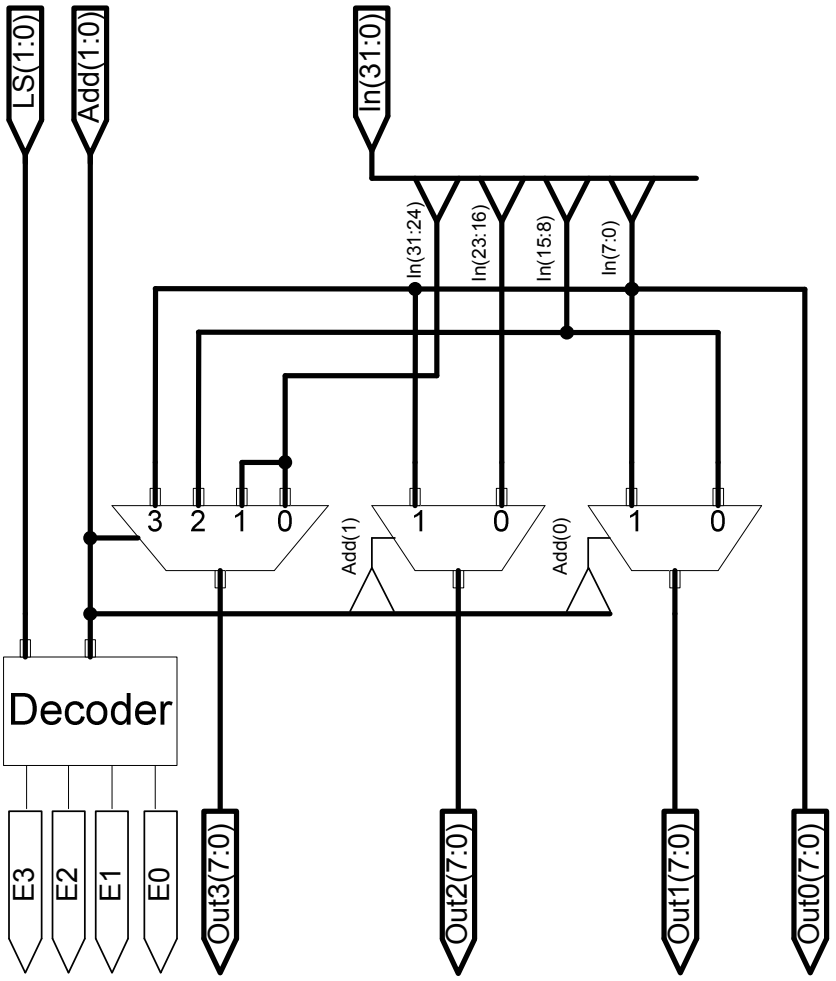
Add(1:0)	LS(1:0)	E3	E2	E1	E0
00	**00**	0	0	0	1
	01	0	0	1	1
	10	1	1	1	1
01	**00**	0	0	1	0
10	**00**	0	1	0	0
	01	1	1	0	0
11	**00**	1	0	0	0

Tabel 3.10 menunjukkan bit-bit dari *register* dan posisinya pada memori data saat penyimpanan.

Tabel 3.10 Posisi Bit pada MCO

Add(1:0)	LS(1:0)	Out3	Out2	Out1	Out0
00	**00**	X	X	X	In(7:0)
	01	X	X	In(15:8)	In(7:0)
	10	In(31:24)	In(23:16)	In(15:8)	In(7:0)
01	**00**	X	X	In(7:0)	X
10	**00**	X	In(7:0)	X	X
	01	In(15:8)	In(7:0)	X	X
11	**00**	In(7:0)	X	X	X

Gambar 3.17 menunjukkan skematik dari *memory control out* (MCO).



Gambar 3.17 Sekamtik *Memory Control Out*

Tabel 3.11 merupakan pemetaan memori data ke *register* dimana angka satu menunjukkan data yang akan dilewatkan dari memori data ke *register*. Tabel ini menunjukkan operasi yang dilakukan oleh *decoder* pada Gambar 3.18.

Tabel 3.11 Dekoder untuk *Enable* pada MCI

Add(1:0)	LS(1:0)	E3	E2	E1	E0
00	00	0	0	0	1
	01	0	0	1	1
	10	1	1	1	1
01	00	0	0	0	1
10	00	0	0	0	1
	01	0	0	1	1
11	00	0	0	0	1

Tabel 3.12 menunjukkan bit-bit dari memori data dan posisinya pada *register* saat pengambilan.

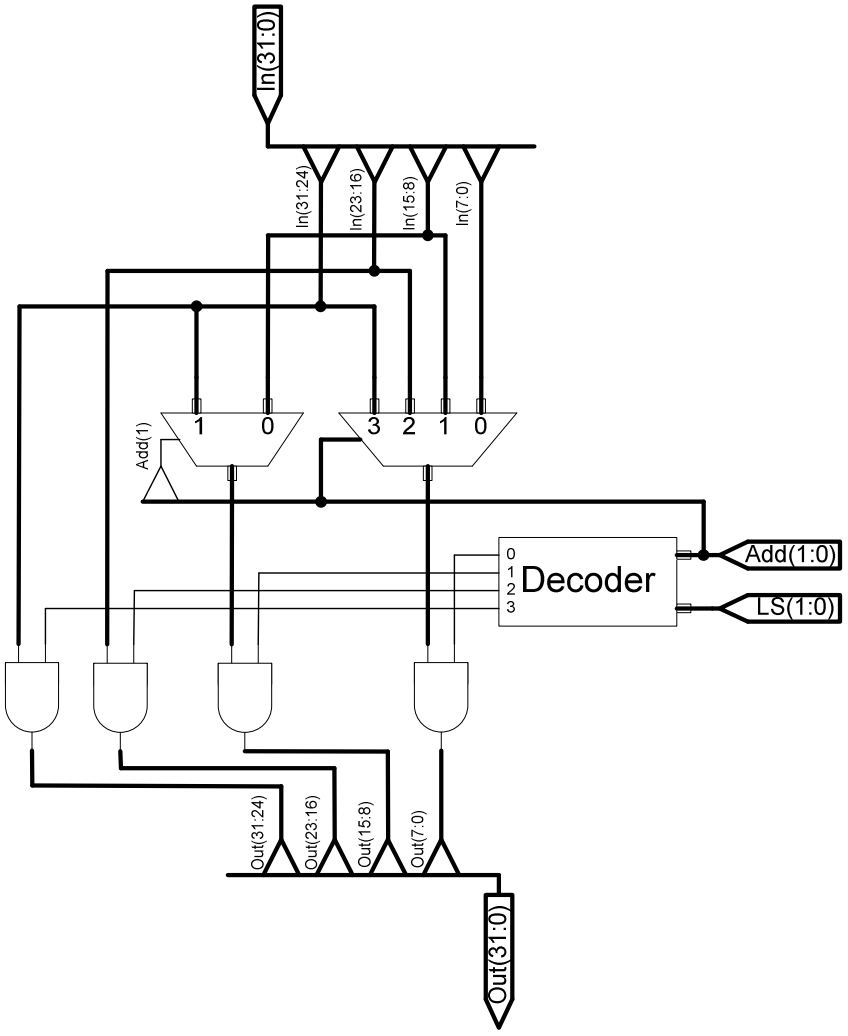
Tabel 3.12 Posisi Bit pada MCI

Add(1:0)	LS(1:0)	Out3	Out2	Out1	Out0
00	00	0	0	0	In(7:0)
	01	0	0	In(15:8)	In(7:0)
	10	In(31:24)	In(23:16)	In(15:8)	In(7:0)
01	00	0	0	0	In(15:8)
10	00	0	0	0	In(23:16)
	01	0	0	In(31:24)	In(23:16)
11	00	0	0	0	In(31:24)

Gambar 3.18 menunjukkan skematik dari *memory control in* (MCI).

3.3.5 *Control Unit*

Control unit berfungsi untuk menentukan instruksi selanjutnya yang akan dieksekusi. Untuk merubah alur program digunakan instruksi percabangan. Instruksi percabangan bersyarat akan dinamakan *branch* dan yang tidak bersyarat dinamakan *jump*.



Gambar 3.18 Skematik *Memory Control In*

3.3.5.1 *Jump*

Perancangan instruksi *jump* (JMP) dapat dilakukan dengan dua cara yaitu *jump* dimana alamatnya targetnya tidak dijumlah dengan PC (*jump point to point*) dan *jump* dimana alamat targetnya dijumlah dengan PC (*jump relative*).

Rentang alamat yang dapat dicapai oleh *jump point to point* adalah dari alamat 0 hingga 64M (26 bit). Untuk penggunaan program

yang pendek *jump point to point* dapat menghemat CLB. Untuk melakukan lompatan relatif dapat digunakan instruksi *branch*.

Sedangkan jangkauan untuk *jump relative* adalah sejauh 32M ke atas atau 32M ke bawah dari alamat keberadaan sekarang. Model ini digunakan untuk program yang berukuran besar dan membutuhkan banyak lompatan relatif dengan jarak yang jauh.

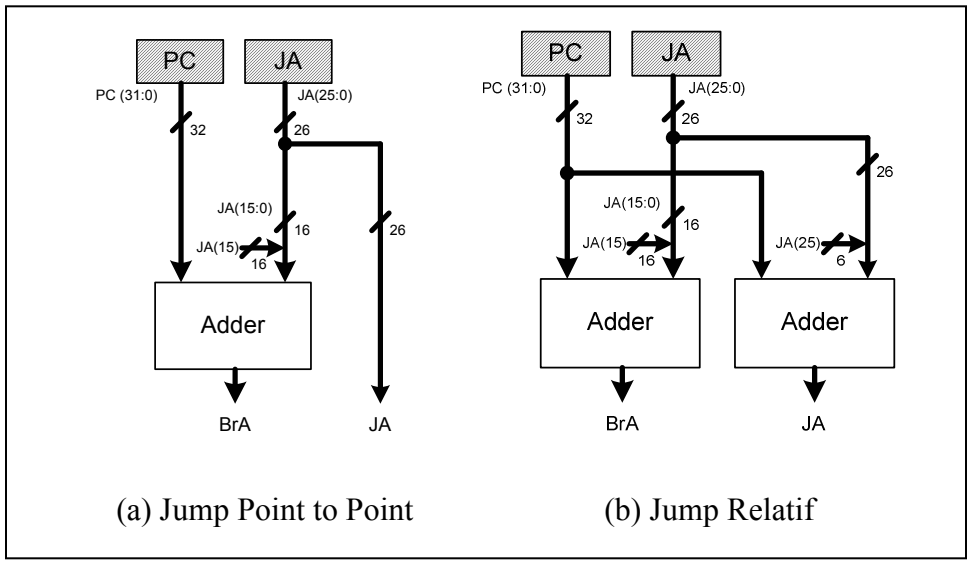
Gambar 3.19 merupakan skematik untuk mengkalkulasi alamat pada operasi *branch* dan *jump*. BrA (*Branch Address*) merupakan alamat untuk operasi *branch* dan JA (*Jump Address*) merupakan alamat untuk operasi *jump*.

Jenis *jump* yang lain adalah *jump and link* (JL) yang digunakan pada instruksi *call*. *Jump* yang digunakan adalah *jump* biasa (*point to point* atau *relative*) tetapi dilakukan bersamaan dengan penyimpanan nilai PC ke *register* yang ditunjuk, yaitu *return address register* (R31). Penyimpanan nilai PC sama seperti yang dilakukan pada instruksi *load address* (LA).

Selain itu terdapat juga *jump register* (JR) dimana alamat target berasal dari nilai *register*. Target dilewatkan melalui *bus* B dengan nama sinyal JRA (*Jump Register Address*). *Jump register* memiliki jumlah jangkauan yang dapat menggapai seluruh alamat memori instruksi. Biasanya JR digunakan untuk instruksi *return* yang mengembalikan nilai *return address register* (R31) ke PC.

Yang terakhir adalah *jump register and link* (JRL). JRL melakukan *jump register* sekaligus menyimpan nilai PC ke *return*

address register (R31). Dimana alamat target berasal dari nilai *register* dan melewati *bus* B sedangkan nilai PC yang ingin disimpan akan melalui *bus* A. Dapat digunakan untuk instruksi *call* dengan jangkauan dapat menggapai seluruh memori instruksi.



Gambar 3.19 Skematik untuk Mengkalkulasi Alamat pada Operasi *Branch* dan *Jump*

3.3.5.2 *Branch*

Branch melakukan percabangan berdasarkan keempat *flag* hasil eksekusi operasi pengurangan pada *function unit* yaitu *carry* (C), *negatif* (N), *overflow* (V), dan *zero*. Percabangan yang dilakukan oleh *branch* adalah relatif terhadap posisi PC. Jangkauan alamat yang dapat digapai adalah 32K ke atas dan 32 K ke bawah dari posisi saat ini.

Ada lima instruksi branch *relational* dan *equality* yang akan dirancang dan dianggap telah mewakili instruksi percabangan *relational* maupun *equality* lainnya. Kelima instruksi branch tersebut adalah *branch*

if equal (BE), *branch if higher* (BH), *branch if higher equal* (BHE), *branch if greater* (BG), dan *branch if greater equal* (BGE).

Branch if equal (BE) akan melakukan percabangan jika kedua *register* yang dibandingkan adalah sama atau *zero flag* yang dihasilkan sama dengan satu.

Branch if higher (BH) dan *branch if higher equal* (BHE) digunakan untuk membandingkan dua *register* yang nilainya merupakan bilangan tidak bertanda. Percabangan dilakukan berdasarkan *carry* dan *zero flag* untuk BH dan *carry flag* saja untuk BHE.

Sedangkan *branch if greater* (BG) dan *branch if greater equal* (BGE) digunakan untuk membandingkan dua *register* yang nilainya merupakan bilangan bertanda. Oleh karena itu percabangan dilakukan berdasarkan *flag* XNV (XOR *negatif* dan *overfloaw*) dan *zero* untuk BG dan XNV saja untuk BGE.

Jenis branch lainnya seperti *branch if zero* dapat dibentuk dengan instruksi *branch if equal* (BE) dengan *zero register* (R0). *Branch if negatif* dapat dilakukan dengan instruksi *branch if greater* (BG) antara *zero register* (R0) dengan nilai tersebut.

Pada operasi penjumlahan dua bilangan tidak bertanda, kesalahan terjadi jika *carry* yang dihasilkan sama dengan satu. Sedangkan untuk pengurangan dua bilangan tidak bertanda, kesalahan terjadi jika *carry* yang dihasilkan sama dengan nol. Maka untuk instruksi penjumlahan dua bilangan tidak bertanda, *branch if carry* dapat dilakukan dengan instruksi *branch if higher* (BH) antara salah satu *register* penjumlahan dengan

register hasil. Jika salah satu *register* penjumlahan lebih besar dari *register* hasil berarti *carry* sama dengan satu. Sedangkan untuk pengurangan dua bilangan tidak bertanda *branch if not carry* dapat dilakukan dengan instruksi *branch if higher* antara hasil pengurangan dengan nilai yang ingin dikurangi. Jika hasil pengurangan lebih besar dari nilai yang ingin dikurangi berarti *carry* sama dengan nol.

Sedangkan solusi untuk instruksi *branch if overflow* adalah dengam menambahkan instruksi ini ke *branch control*. Agar *branch selection* efisien dan *branch if overflow* dapat dimanipulasi dengan memasukkan kedua nilai operasi ke dalam register maka pada perancangan *branch control* tidak digunakan *branch if overflow*.

3.3.5.3 *Branch Control*

Untuk melakukan percabangan maka nilai PC akan diambil dari berbagai alamat percabangan. Karena itu dibutuhkan sebuah *multiplexer* yang akan memilih alamat selanjutnya yang akan diambil berdasarkan sinyal yang berasal dari *branch kontrol*, *multiplexer* ini disebut MUX C. Tabel 3.13 merupakan tabel untuk MUX C.

Tabel 3.13 *Multiplexer* C

MC	Operation
00	PC ← PC + 1
01	PC ← BrA
10	PC ← JA
11	PC ← JRA

Komponen *branch kontrol* berfungsi untuk menentukan apakah terjadi percabangan atau tidak, jika terjadi percabangan maka jenis percabangan apa yang diinginkan. Sinyal yang melakukan hal ini disebut *branch selector* (BS). Tabel 3.14 menunjukkan jenis-jenis percabangan, operasi yang dilakukannya, *flag* yang berpengaruh, masukan berupa sinyal BS untuk menentukan jenis percabangan, dan keluaran berupa sinyal MC untuk mengontrol MUX C. Gambar 3.20 menunjukkan skematik *control unit*.

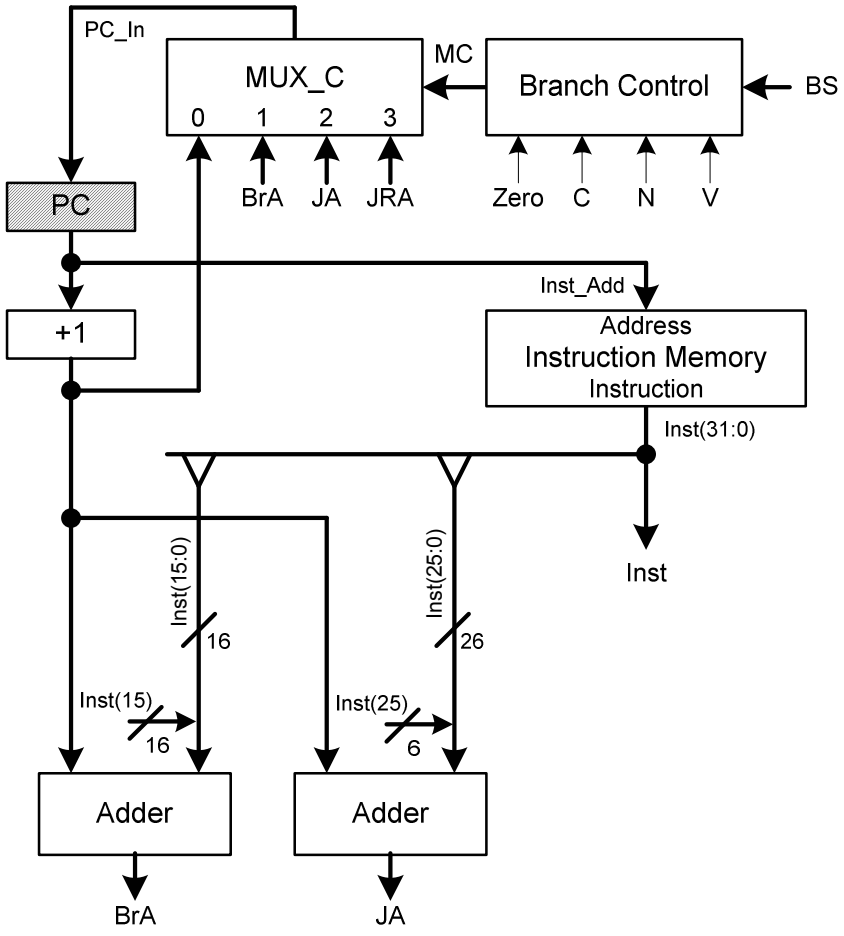
Tabel 3.14 Operasi Percabangan

BS	Flag	MC	Operation	Instruction Name
000	C=1 AND Zero=0	01	PC ← PC+Imm ← BrA	Branch if Higher
000	C=0 OR Zero=1	00	PC ← PC+1	
001	C=1	01	PC ← PC+Imm ← BrA	Branch if Higer Equal
001	C=0	00	PC ← PC+1	
010	Zero=1	01	PC ← PC+Imm ← BrA	Branch if Equal
010	Zero=0	00	PC ← PC+1	
011	XNV=0 AND Zero=0	01	PC ← PC+Imm ← BrA	Branch if Greater
011	XNV=1 OR Zero=1	00	PC ← PC+1	
100	XNV=0	01	PC ← PC+Imm ← BrA	Branch if Greater Equal
100	XNV=1	00	PC ← PC+1	
101	---	10	PC ← PC+Target ← JA	Jump
110	---	11	PC ← RB ← JRA	Jump Register
111	---	00	PC ← PC+1	Next

3.3.6 *Instruction Decoder*

Pada perancangan *instruction decoder*, *Opcode* diharapkan untuk disusun sedemikian rupa sehingga jumlah *slice* yang digunakan dan TPD yang dihasilkan sekecil mungkin. Beberapa organisasi pada prosesor RISC menempatkan sebagian *Opcode* pada *unuse* bit untuk format instruksi *register*. Dengan jumlah bit *Opcode* yang diperbanyak diharapkan kerumitan dalam mendekode instruksi dapat dihindari sehingga *slice* dan TPD yang

dihasilkan menjadi lebih kecil. Perancangan yang demikian tidak berlaku jika implementasi yang dilakukan menggunakan FPGA. LUT sangat bergantung dari jumlah *input* yang diberikan, sehingga jika digunakan lebih dari 6 bit *Opcode* maka hasil yang terjadi adalah semakin banyak LUT yang dibutuhkan dan penumpukan LUT akan mengakibatkan TPD semakin membesar. Berdasarkan karakteristik tersebut maka *Opcode* yang dirancang hanya diurutkan berdasarkan pengelompokan jenis operasinya.



Gambar 3.20 Skematik *Control Unit*

Instruction decoder melakukan dekode *Opcode* menjadi sinyal-sinyal yang berfungsi untuk mengontrol jalannya operasi baik pada *data path* maupun pada *control unit*. Kumpulan sinyal-sinyal kontrol ini disebut *control word*. *Control word* terdiri dari 15 bit dan dibagi menjadi 9 bagian dimana fungsi masing-masingnya dapat dilihat pada Tabel 3.15.

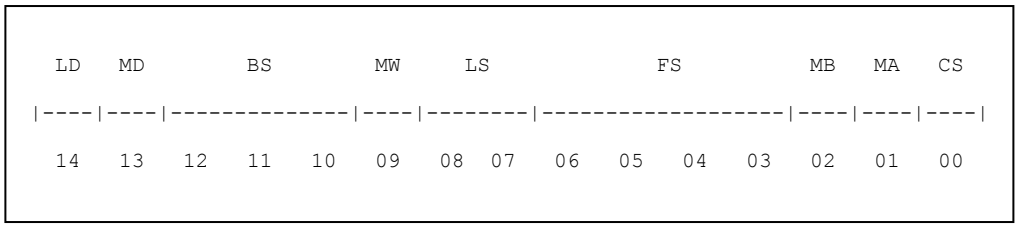
Tabel 3.15 Sinyal-sinyal Kontrol

Signal	Code	Function
Load (LD)	0	Tidak Tulis ke Register File
	1	Tulis ke Register File
Multiplexer D (MD)	0	Function Unit
	1	Memori Data
Branch Select (BS)	000	Branch if Higher
	001	Branch if Higher Equal
	010	Branch if Equal
	011	Branch if Greater
	100	Branch if Greater Equal
	101	Jump
	110	Jump Register
	111	Next Instruction
Memory Write (MW)	0	Tidak Tulis ke Memori Data
	1	Tulis ke Memori Data
Load/Store (LS)	00	Load/ Store Byte
	01	Load/ Store Half Word
	10	Load/ Store Word
	11	(X)
Function Select (FS)	0000	Addition
	0001	Subtraction
	0010	Pass
	0011	Pass (X)
	0100	AND
	0101	OR
	0110	XOR
	0111	NOR
	1000	Shift Logical Right
	1001	Shift Logical Left
	1010	Shift Arithmetic Right
	1011	(X)

Tabel 3.15 Sinyal-sinyal Kontrol (lanjutan)

Signal	Code	Function
Function Select (FS)	1100	(X)
	1101	Set
	1110	(X)
	1111	LUI
Multiplexer B (MB)	0	Register File
	1	Immediate
Multiplexer A (MA)	0	Register File
	1	PC
Constant Selct (CS)	0	Bilangan Tidak Bertanda
	1	Bilangan Bertanda

Gambar 3.21 menunjukkan posisi *bit* pada *control word* yang merupakan hasil dekode dari *Opcode* dan digunakan untuk mengontorl apa yang harus dilakukan oleh komponen-komponen yang membangun baik *datapath* maupun *control unit*.



Gambar 3.21 Posisi Bit pada *Control Word*

Tabel 3.16 menunjukkan *mnemonic*, *Opcode*, *operand*, nilai *control word*, dan format instruksi dari masing-masing instruksi. Pada kolom format instruksi, R berarti format *register*, I berarti format *immediate*, B berarti format *branch*, dan J berarti format *jump*.

Tabel 3.16 *Instruction Decoder*

Instruxtion	Mnemonic/Opcode		Operand	Control Word Values	IF
Addition	ADD	000000	RD, RA, RB	1.0.111.0.00.0000.0.0.0	R
Addition Immediate Sign	ADI	000001	RD, RA, Imm	1.0.111.0.00.0000.1.0.1	I
Addition Immediate Unsign	ADIU	000010	RD, RA, Imm	1.0.111.0.00.0000.1.0.0	I
Subtract	SUB	000011	RD, RA, RB	1.0.111.0.00.0001.0.0.0	R
Subtract Immediate Sign	SBI	000100	RD, RA, Imm	1.0.111.0.00.0001.1.0.1	I
Subtract Immediate Unsign	SBIU	000101	RD, RA, Imm	1.0.111.0.00.0001.1.0.0	I
AND	AND	000110	RD, RA, RB	1.0.111.0.00.0100.0.0.0	R
AND Immediate	ANDI	000111	RD, RA, Imm	1.0.111.0.00.0100.1.0.0	I
OR	OR	001000	RD, RA, RB	1.0.111.0.00.0101.0.0.0	R
OR Immediate	ORI	001001	RD, RA, Imm	1.0.111.0.00.0101.1.0.0	I
XOR	XOR	001010	RD, RA, RB	1.0.111.0.00.0110.0.0.0	R
XOR Immediate	XORI	001011	RD, RA, Imm	1.0.111.0.00.0110.1.0.0	I
NOR	NOR	001100	RD, RA, RB	1.0.111.0.00.0111.0.0.0	R
NOR Immediate	NORI	001101	RD, RA, Imm	1.0.111.0.00.0111.1.0.0	I
Shift Logical Right	SLR	001110	RD, RA, Imm	1.0.111.0.00.1000.1.0.0	I
Shift Logical Right Variable	SLRV	001111	RD, RA, RB	1.0.111.0.00.1000.0.0.0	R
Shift Logical Left	SLL	010000	RD, RA, Imm	1.0.111.0.00.1001.1.0.0	I
Shift Logical Left Variable	SLLV	010001	RD, RA, RB	1.0.111.0.00.1001.0.0.0	R
Shift Arithmetic Right	SAR	010010	RD, RA, Imm	1.0.111.0.00.1010.1.0.0	I
Shift Arithmetic Right Variable	SARV	010011	RD, RA, RB	1.0.111.0.00.1010.0.0.0	R
Load Upper Immediate	LUI	010100	RD, Imm	1.0.111.0.00.1111.1.0.0	I
Load Address	LA	010101	RD	1.0.111.0.00.0010.0.1.0	R
Store Byte	SB	010110	RA, RB	0.0.111.1.00.0010.0.0.0	R
Store Half Word	SH	010111	RA, RB	0.0.111.1.01.0010.0.0.0	R
Store Word	SW	011000	RA, RB	0.0.111.1.10.0010.0.0.0	R
Load Byte	LB	011001	RD, RA	1.1.111.0.00.0010.0.0.0	R
Load Half Word	LH	011010	RD, RA	1.1.111.0.01.0010.0.0.0	R
Load Word	LW	011011	RD, RA	1.1.111.0.10.0010.0.0.0	R
Set if Less Then	SLT	011100	RD, RA, RB	1.0.111.0.00.1101.0.0.0	R
Set if Less Then Immdeiate Sign	SLTI	011101	RD, RA, Imm	1.0.111.0.00.1101.1.0.1	I
Disable Interrupt	DI	011110	--	0.0.111.0.00.0010.0.0.0	-
Enable Interrupt	EI	011111	--	0.0.111.0.00.0010.0.0.0	-
Branch if Equal	BE	100000	RA, RB, Imm	0.0.010.0.00.0001.0.0.0	B
Branch if Higher	BH	100001	RA, RB, Imm	0.0.000.0.00.0001.0.0.0	B
Branch if Higher Equal	BHE	100010	RA, RB, Imm	0.0.001.0.00.0001.0.0.0	B
Branch if Greater	BG	100011	RA, RB, Imm	0.0.011.0.00.0001.0.0.0	B
Branch if Greater Equal	BGE	100100	RA, RB, Imm	0.0.100.0.00.0001.0.0.0	B
Jump	JMP	100101	Target	0.0.101.0.00.0010.0.0.0	J
Jump and Link	JL	100110	Target	1.0.101.0.00.0010.0.1.0	J
Jump Register	JR	100111	RB	0.0.110.0.00.0010.0.0.0	R
Jump Register and Link	JRL	101000	RB	1.0.110.0.00.0010.0.1.0	R

3.3.7 *Pipeline*

Pada perancangan *pipeline*, operasi prosesor dibagi menjadi empat tahap yaitu *instruction fetch* (IF), *instruction decoder* dan *operand fetch* (DO), *execution* (EX), dan *write back* (WB). Tahap IF melakukan penggambilan instruksi dari memori instruksi dan menambah nilai PC dengan satu untuk mengambil instruksi berikutnya. Tahap DO melakukan dekode instruksi dan pengambilan nilai *operand* berdasarkan jenis instruksinya. Tahap EX melakukan eksekusi untuk beberapa jenis instruksi, melakukan persiapan target alamat untuk instruksi percabangan, dan melakukan pengambildan/penyimpanan data dari/ke memori data untuk instruksi *load*/*store*. Tahap WB melakukan penyimpanan data ke *register file*.

Tabel 3.17 menunjukkan tahap *pipeline* berdasarkan pengelompokan jenis instruksi. InstMem menunjukkan memori instruksi (*instruction memory*), DataMem menunjukkan data memori, Reg menunjukkan *register*, Imm menunjukkan *immediate*, op menunjukkan *operation*, dan se menunjukkan *sign extension* yang merupakan perluasan dari bit ke 15 pada bilangan bertanda. Gambar 3.22 merupakan skematik prosesor dengan *pipeline*.

Beberapa perancangan untuk menghindari *data* dan *control hazard* yaitu menggunakan *data hazard stall*, *data forwarding* dan *branch prediction*.

Tabel 3.17 Operasi Instruksi Berdasarkan Tahap *Pipeline*

Step	Instruction Type					
	Register Type	**Immediate Type**	**Load**	**Store**	**Branch Type**	**Jump Type**
IF	IR ← InstMem[PC]; PC ← PC + 1;					
DO	SA ← Reg[IR(20:16)]; SB ← Reg[IR(15:11)];	SA ← Reg[IR(20:16)]; Imm ← se || IR(15:0);	SA ← Reg[IR(20:16)];	SA ← Reg[IR(20:16)]; SB ← Reg[IR(15:11)];	JA ← se || IR(25:21) || IR(10:0);	JA ← se || IR(25:0);
EX	DR ← SA op SB;	DR ← SA op Imm;	DR ← DataMem[SA];	DataMem[SA] ← SB;	If (TRUE) then PC ← PC_2 + JA;	PC ← PC_2 + JA;
WB	Reg[IR(25:21)] ← DR					

3.3.7.1 *Data Hazard Stall*

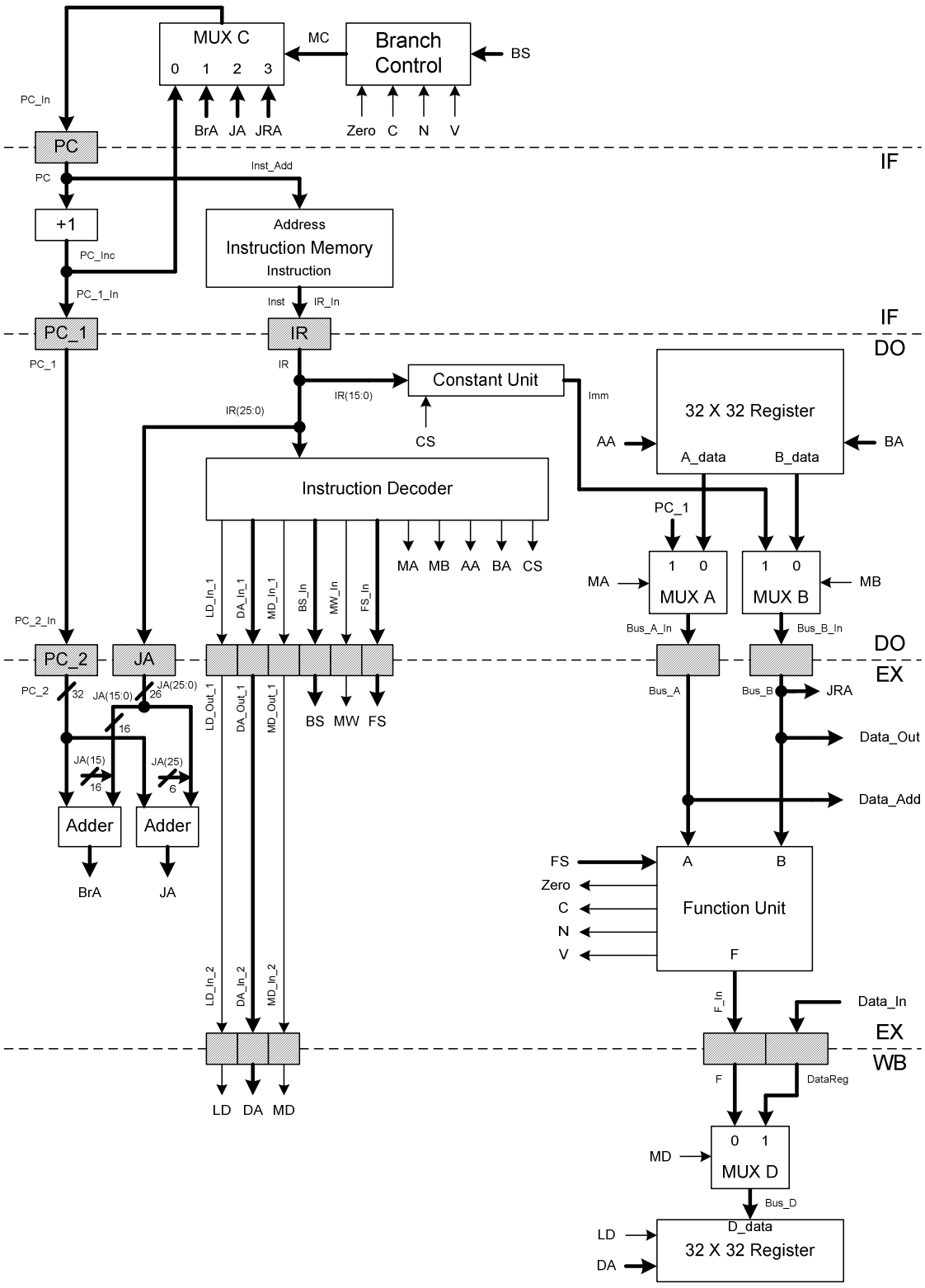
Solusi *data hazard stall* akan membandingkan apakah DA_{EX} sama dengan AA_{DO} atau BA_{DO}, apakah terjadi penulisan ke *register* ($LD_{EX} = 1$) dan bukan ke *register* nol (DA_{EX} /= 00000) serta apakah operasi berikutnya yang diambil berasal dari *register* ($MA_{DO} = 0$ atau $MB_{DO} = 0$). Berikut adalah persamaan matematisnya:

$$HA = (DA_{EX} = AA_{DO}) \,.\, LD_{EX} \,.\, \sum_{i=0}^{4} (DA_{EX})_i \,.\, \overline{MA_{DO}}$$

$$HB = (DA_{EX} = BA_{DO}) \,.\, LD_{EX} \,.\, \sum_{i=0}^{4} (DA_{EX})_i \,.\, \overline{MB_{DO}}$$

$$DD = HA + HB$$

Jika DD sama dengan satu berarti terjadi *data depedency* maka PC tidak ditambah dengan satu dan IR tidak mengambil instruksi berikutnya. *Control word* yang dapat merubah nilai *register* (LD_{DO}) dan memori (MW_{DO}) maupun merubah aliran program (BS_{DO}) harus berlaku seperti instruksi NOP.



Gambar 3.22 Skematik Prosesor dengan *Pipeline*

3.3.7.2 *Data Forwarding*

Pada *data forwarding* dilakukan pemeriksaan *data depedency* seperti yang dilakukan pada *data hazard stall*. Jika ternyata terjadi *data dependency* maka *bus* data keluaran dari *function unit* dan memori data yang menuju MUX D dicabangkan juga ke MUX_D_DF yang di kontrol oleh MD_{EX}. Data hasil MUX_D_DF langsung memasuki MUX A atau MUX B yang dikontrol oleh HA atau HB untuk melalui Bus A atau Bus B tanpa menemui *register* penampung dan dapat langsung digunakan sebagai masukkan untuk instruksi selanjutnya. Perancangan ini mengakibatkan MUX A dan MUX B memiliki dua bit selektor dengan tiga jenis input.

3.3.7.3 *Branch Prediction*

Branch predeiction akan membandingkan MC_{EX} yang digunakan untuk percabangan (*branch* dan *jump*). Percabangan terjadi jika MC_{EX} tidak sama dengan nol. Oleh karena itu *control word* yang berpengaruh terhadap perubahan nilai *register* (LD_{DO}) dan memori (ME_{DO}), kemungkinan terjadinya percabagan (BS_{DO}) dan pemgambilan instruksi pada tahap IF harus diperhatikan. Hal ini dikarenakan *branch hazard* mengambil dua tahap *pileline* sebelum menentukan apakah terjadi percabangan. Dua tahap tersebut tidak langsung di beri *bubble* melainkan menunggu hingga percabangan terjadi. Keuntungannya adalah jika percabangan tidak terjadi maka dua instruksi selanjutnya yang telah di ambil dan didekode dapat dilanjutkan.

3.3.8 *Interrupt*

Untuk menghindari sistem *poolling* dimana prosesor harus memeriksa secara terus-menerus apakah ada peralatan luar yang meminta layanan maka dirancang sistem *interrupt* dimana prosesor mendapat sinyal masukan berupa permintaan *interrupt* (*interrupt request*) dari peralatan luar. Permintaan *interrupt* disertai dengan alamat yang memberitahukan peralatan mana yang meminta layanan dan proses apa yang harus dilakukan oleh prosesor. Alamat ini disebut alamat *interrupt* (*interrupt address*). Pada perancangan, alamat *interrupt* masuk melalui jalur alamat data (*data address*) sehingga *interrupt* tidak boleh terjadi bersamaan dengan penulisan atau pembacaan data ke atau dari memori data atau peralatan luar.

Untuk membatasi kapan *interrupt* boleh terjadi dan kapan tidak maka dirancang instruksi yang dapat memberitahukan keadaan tersebut. Instruksi ini adalah *enable interrupt* (EI) dan *disable interrupt* (DI). Jika permintaan *interrupt* diijinkan maka prosesor akan memberikan sinyal berupa *interrupt acknowledge* ke peralatan yang meminta interrupt.

Yang dilakukan oleh prosesor pada saat interrupt terjadi:

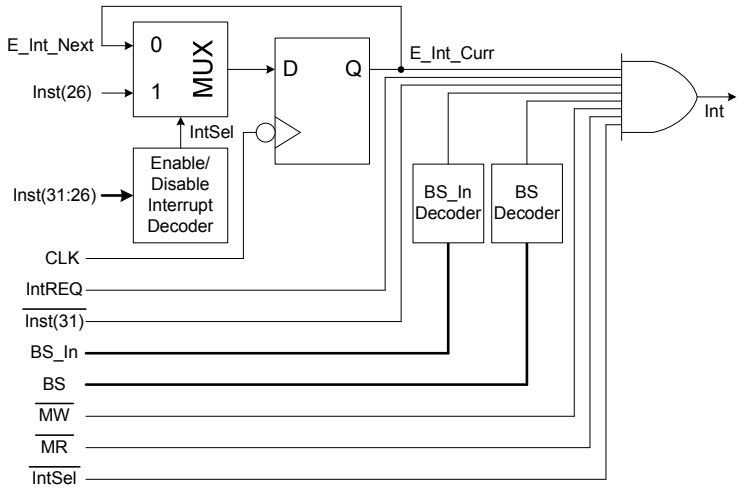
1. Memberikan sinyal *interrupt acknowledge* ke peralatan yang meminta *interrupt*.
2. PC diisi dengan *interrupt address* (IntADD) untuk menuju *interrupt vector table*. MUX_I_PC digunakan untuk memilih apakah *interrupt address* atau instruksi selanjutnya (dari MUX C) yang akan menjadi nilai PC yang baru.

3. PC_1 diisi dengan nilai PC bukan PC + 1 karena pada saat *interrupt* terjadi instruksi yang dikerjakan adalah instruksi *load address* ke *return address interrupt register* (R30) dan *load address* harus menyimpan alamat saat ini bukan alamat berikutnya. *Multiplexer* yang memilih antara PC dan PC + 1 untuk nilai PC_1 disebut MUX_I_PC_1.
4. *Instruction register* (IR) akan mengambil instruksi *load address* dengan *register* yang dituju adalah *return address interrupt register* (R30). *Multiplexer* yang memilih antara instruksi dari memori instruksi atau instruksi *load address* disebut MUX_IR.

Pada beberapa situasi, *interrupt* tidak boleh terjadi karena akan menyebabkan terjadinya kesalahan pada program yang sedang dijalankan. Oleh karena itu interrupt hanya boleh terjadi jika:

1. Ada permintaan *interrupt*.
2. *Interrupt* dalam keadaan *enable*.
3. Tahap IF tidak sedang mengambil operasi percabangan dan operasi *interrupt*.
4. Tahap DO tidak sedang mendekode operasi percabangan.
5. Tahap EX tidak sedang mengeksekusi operasi percabangan.
6. Tidak sedang melakukan penulisan atau pembacaan ke atau dari memori data atau peralatan luar.

Komponen untuk mengatur apakah *interrupt* boleh terjadi atau tidak disebut *interrupt control*. Gambar 3.23 merupakan skematik *interrupt control*.



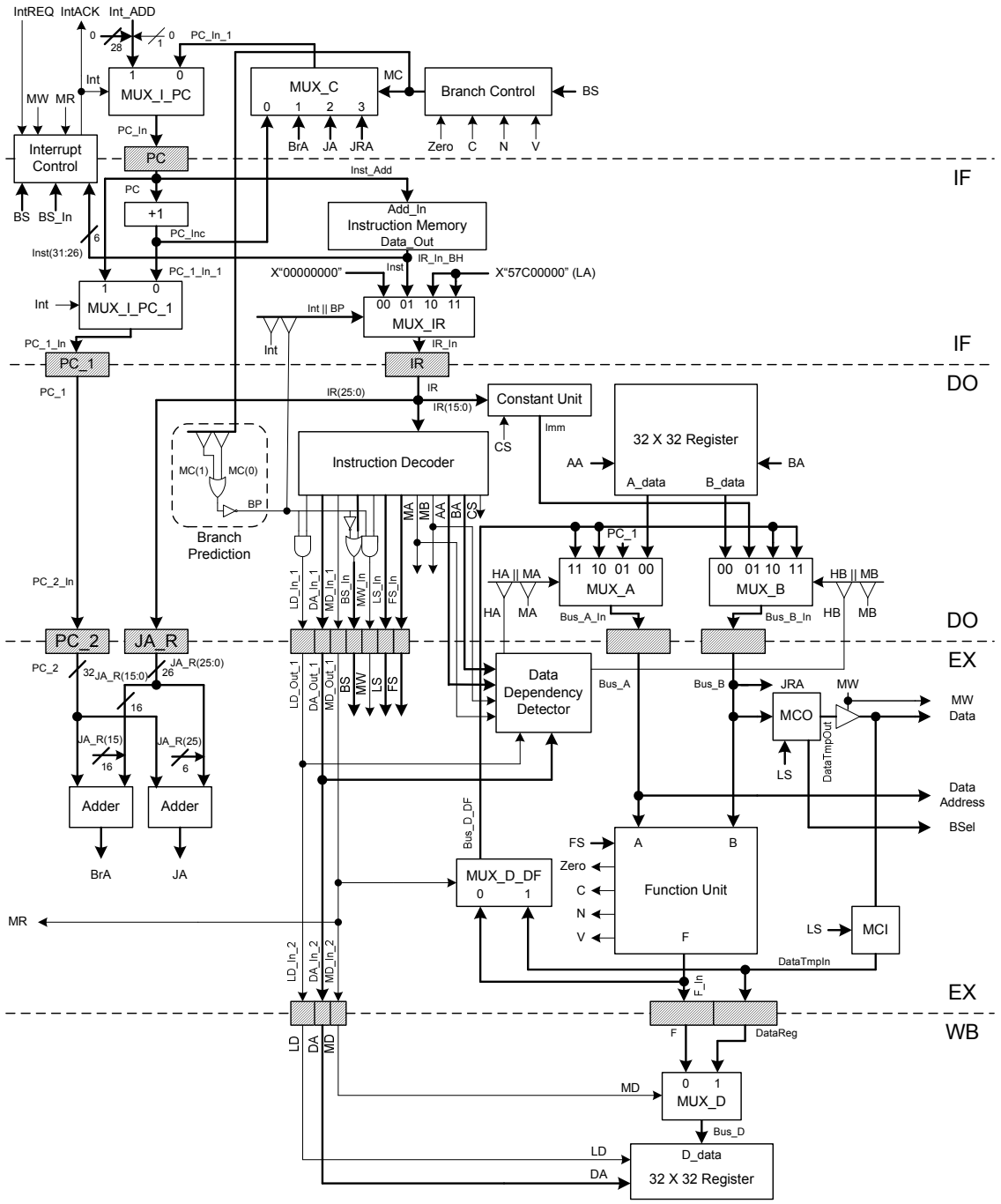
Gambar 3.23 Skematik *Interrupt Control*

Sinyal Inst(31:26) akan didekode untuk mengetahui apakah instruksinya merupakan instruksi *interrupt*. Jika ya maka IntSel akan mengambil nilai dari Inst(26) yang bernilai 1 untuk instruksi EI dan 0 untuk instruksi DI. Jika tidak maka sinyal yang akan dilewatkan adalah sinyal E_Int saat ini. Sinyal E_Int yang menentukan apakah *interrupt* boleh terjadi atau tidak disamping persyaratan lain untuk terjadinya *interrupt*.

Sinyal Inst(31) yang menandakan apakah instruksi percabangan yang diambil, sinyal BS_In yang menandakan apakah instruksi percabangan yang di decode, dan sinyal BS yang menandakan apakah instruksi percabangan yang dieksekusi.

Gambar 3.24 merupakan skematik dari prosesor dengan arsitektur RISC yang dirancang. Untuk mengatasi *data hazard* dan *control hazard* digunakan *data forwarding* dan *branch control*. Digunakan *memory control in* (MCI) dan

memory control out (MCO) untuk mengatur data yang masuk dan keluar ke dan dari prosesor. Agar dapat menangani *interrupt* maka dimasukkan juga komponen *interrupt* dan *interrupt control*.



Gambar 3.24 Skematik Prosesor RISC