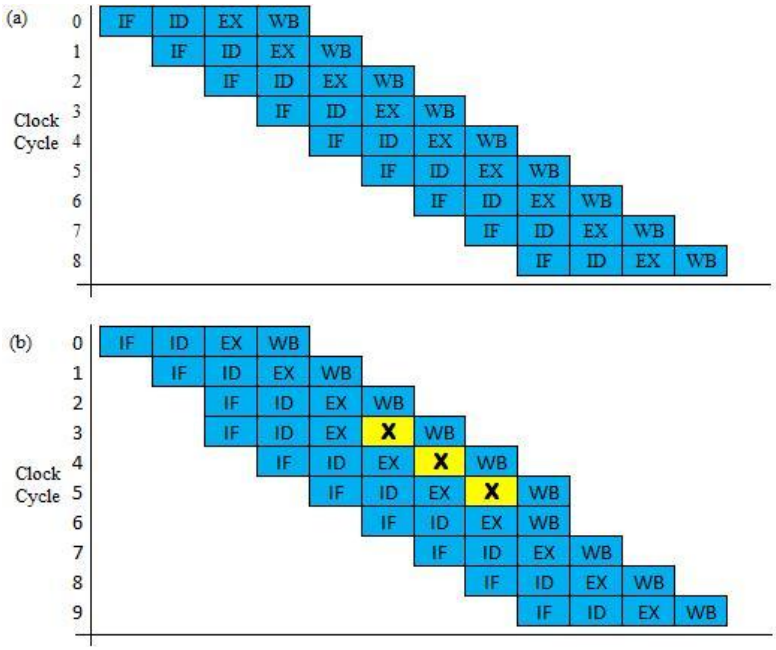
BAB II

DASAR TEORI

2.1 *Branch Prediction*

Branch prediction adalah sebuah metode dimana sebuah prosesor menebak hasil dari sebuah *branch instruction* agar prosesor tersebut dapat bersiap untuk melaksanakan instruksi sesuai dengan hasil yang telah diperkirakan sebelumnya.

Tanpa *branch prediction* yang akurat, kinerja prosesor akan terhambat, karena *branch instruction* dapat menyebabkan *delay* dalam alur *pipeline* prosesor. Fenomena ini disebut dengan istilah "*bubble*", yaitu hambatan dimana tidak terjadi apa-apa dalam proses *fetch*, *decode*, *execute*, dan *writeback* pada *pipeline*.



Gambar 2.1 Ilustrasi *pipeline* dengan (a) eksekusi normal dan (b) eksekusi dengan *bubble*[1]

Seperti yang terlihat dalam gambar, karena terjadi *delay* dalam mengambil keputusan pada *branch instruction*, pada *cycle* kedua proses *fetch*ing dari instruksi yang berwarna ungu terhambat, dan *decoding stage* pada *cycle* ketiga berisi *bubble*.

Instruksi-instruksi lain yang ada “dibelakang” instruksi *execute* (EX) juga menjadi terhambat, namun instruksi yang ada “didepan” instruksi *execute* (EX) berjalan seperti biasa. Dapat dilihat jika dibandingkan dengan eksekusi instruksi normal, *bubble* menyebabkan total waktu eksekusi menjadi 8 *clock tick* dari yang seharusnya 7 *clock tick*.

Prosesor-prosesor modern memiliki *pipeline* yang panjang, masalah dari *pipeline* panjang adalah apabila sebuah program bercabang (memiliki *branch instruction*), prosesor tidak dapat mengetahui dimana dia harus mengambil instruksi selanjutnya dan harus menunggu sampai *branch instruction* selesai, sehingga menyebabkan *pipeline* dibelakangnya kosong. Masalah ini bisa dikurangi dengan *branch prediction*.

Dalam eksekusi instruksi pada komputer, memprediksikan hasil dari sebuah *branch instruction* agar instruksi-instruksi tersebut nantinya dapat dijalankan secara paralel dengan instruksi-instruksi yang sedang dijalankan saat itu. Apabila komputer menebak *branch* yang salah, akan membutuhkan machine *cycle* yang lebih untuk kembali dan menjalankan *branch* yang benar; namun secara rata-rata, apabila algoritma *prediction*nya bagus, performa secara keseluruhan menjadi lebih bagus.

Stallings[3] mendeskripsikan cara kerja teknik *branch predictor*, yaitu prosessor melihat kode instruksi selanjutnya dari memori, kemudian memprediksi percabangan atau kelompok instruksi yang mirip untuk diproses berikutnya. Apabila perkiraan prosessor benar pada bebarapa waktu tertentu, prosessor akan mengambil instruksi-instruksi yang benar dan menyimpannya di dalam *buffer*, sehingga prosessor selalu dalam keadaan sibuk. Prediksi *branch predictors* tidak hanya pada sebuah percabangan selanjutnya, tetapi juga beberapa cabang berikutnya.

Penelitian *branch prediction* untuk mendukung *performance* prosessor *modern* dalam menangani percabangan instruksi telah banyak dilakukan. *Dynamic branch predictor* yang pertama untuk mengambil prediksi percabangan didasarkan pada *history* informasi lokal. Sejak itu, *branch predictor* mengalami perkembangan yang

signifikan. Perkembangan *branch predictor* ditentukan diantaranya oleh 3 (tiga) kategori dasar:

1. penambahan *path global* dan *history* informasi
2. teknik mengkombinasikan antara *history global* dan lokal, dan
3. mengurangi hambatan melalui skema indeks tabel yang lebih baik.

2.1.1 Teknik *Branch Prediction*

Dua komponen fundamental dari *branch prediction* adalah *branch target Speculation* dan *branch Condition Speculation*.

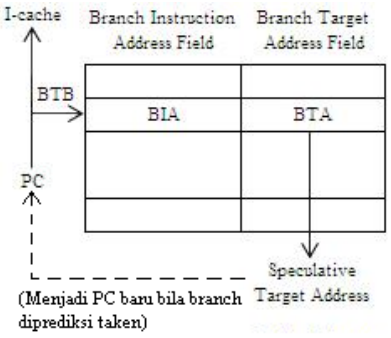
2.1.1.1 *Branch Target Speculation*

Branch target Speculation melibatkan pemakaian *Branch Target Buffer* (BTB) untuk menyimpan *Branch Target Address* (BTA) yang sebelum-sebelumnya. BTB adalah sebuah *memory cache* yang diakses pada tahap *instruction fetch* dengan *instruction fetch address/program counter* (PC). Tiap *entry* pada BTB memiliki 2 kolom, yaitu kolom *Branch Intruction Address* (BIA) dan kolom BTA.

Saat sebuah *static branch instruction* dijalankan pertama kali, *entry* dari BTB dialokasikan untuk instruksi tersebut. *Address* dari instruksi tadi disimpan dalam kolom BIA, dan *target address*nya disimpan di kolom BTA.

BTB diakses bersamaan dengan akses pada I-*cache*. Jika PC yang bersangkutan cocok dengan BIA dari *entry* pada BTB, terjadi "*hit*" pada BTB. Ini menunjukkan bahwa instruksi yang diambil dari I-*cache* pada saat itu telah dijalankan sebelumnya dan merupakan *branch instruction*.

Bila terjadi "*hit*" pada BTB, BTA dari "*hit*" *entry* diakses dan dapat dipakai sebagai PC berikutnya jika *branch instruction* bersangkutan diprediksi sebagai *taken*. Ilustrasi dari *branch target Speculation* dengan *branch target buffer* dapat dilihat pada gambar berikut.



Gambar 2.2 *Branch Target Speculation* dengan *Branch Target Buffer*[1]

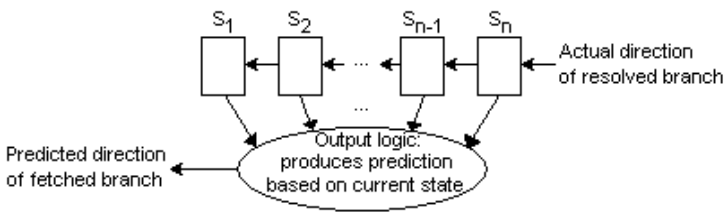
2.1.1.2 *Branch Condition Speculation*

Ada beberapa cara untuk melakukan *Branch Condition Speculation*. Yang paling mudah adalah dengan cara mendesign *fetch hardware* agar selalu predict *not-taken*. Bila *branch instruction* ditemukan, tahap *fetch* akan terus melakukan pengambilan data tanpa terhambat.

Bentuk minimal dari *branch prediction* ini mudah diimplementasi, namun tidak begitu efektif, karena beberapa *branch* dipakai sebagai instruksi pengakhir *loop*, yang biasanya menjadi *taken* pada saat dijalankan kecuali ketika keluar dari *loop*.

Teknik *branch Condition Speculation* yang paling umum adalah spekulasi yang didasarkan dengan *history* dari *branch execution* sebelum-sebelumnya. *History-based branch prediction* melakukan prediksi berdasarkan pengamatan dari arah *branch* sebelumnya. Perkiraan ini didapat dari asumsi bahwa informasi historis dari arah yang diambil *static branch* pada eksekusi sebelum-sebelumnya dapat memberi petunjuk arah yang kira-kira akan diambil pada eksekusi-eksekusi selanjutnya.

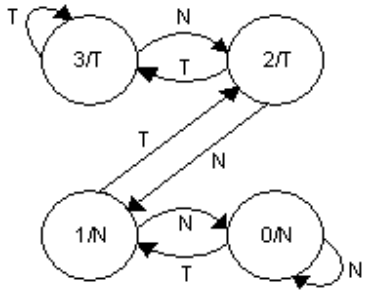
Algoritma spesifik untuk *history-based branch prediction* dapat dikarakterisasikan dengan *Finite State Machine* (FSM).



Gambar 2.3 Model FSM untuk *History*-based *Branch* Direction *Predictor*[1]

Variabel *state n* meng*encode* arah yang diambil dari eksekusi *n* terakhir pada *branch* yang bersangkutan, maka setiap *state* mewakili *history pattern* tertentu dalam pernyataan *taken* dan *not-taken*nya. *Output logic* menghasilkan prediksi yang didasari pada *state* FSM pada saat itu. Kemudian prediksi dibuat berdasarkan *outcome* dari eksekusi-eksekusi *n* sebelumnya dari *branch*. Setelah *branch* yang diprediksi telah dieksekusi, *outcome* sebenarnya dipakai sebagai *input* ke FSM untuk memicu transisi *state.*

State logic berikutnya didapat dengan menggabungkan variabel *state* yang ada menjadi *shift register*, yang menyimpan *branch direction* dari eksekusi-eksekusi *n* sebelumnya dari *branch instruction* tersebut.

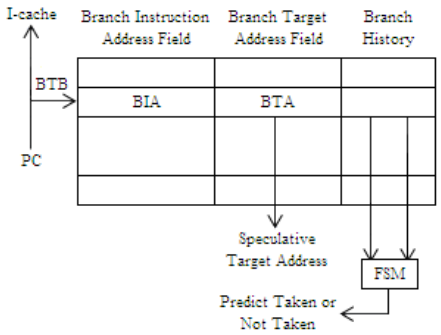


Gambar 2.4 Model algoritma 2-Bit *Branch Predictor*[11]

Gambar 2.4 mengilustrasikan diagram FSM dari 2-bit *branch predictor* yang memakai dua *history bit* untuk melacak hasil dari dua eksekusi sebelumnya dari *branch* tersebut. *Predictor* memiliki 4 *state* yang berbeda: *Strongly Not-taken* (NN), *Weakly Not-taken* (NT), *Strongly Taken* (TT), atau *Weakly Taken* (TN), yang mewakili arah yang diambil pada dua eksekusi sebelumnya pada *branch*. *State* NN dapat dianggap sebagai *state* awal.

Nilai *output* T atau N mewakili prediksi yang akan diambil ketika prediksi berada pada *state* yang bersangkutan. Saat *branch* dieksekusi, arah sebenarnya yang diambil dipakai sebagai *input* pada FSM, dan transisi *state* terjadi untuk mengupdate *branch history* yang akan digunakan pada prediksi selanjutnya.

Untuk mendukung *history-based branch direction predictor*, BTB dapat ditambahkan dengan kolom *Branch History* untuk setiap *entry*nya. Lebar kolom ini (dalam bit) ditentukan dengan jumlah *history* bit yang dilacak. Ketika PC *address* mengalami "*hit*" pada BTB, *history bit* diambil dari kolom *Branch History*. *History bit* ini dimasukkan ke dalam sistem *logic* yang mengimplementasikan *next-state* dan fungsi *output* dari *branch predictor* FSM. Ilustrasi BTB ini dapat dilihat pada gambar dibawah



Gambar 2.5 BTB dengan kolom tambahan untuk menyimpan *Branch History* Bit[1]

2.1.2 Algoritma Dasar Dalam *Branch Prediction*

Secara umum, *branch prediction* pada prosesor memiliki 2 teknik algoritma dasar, yaitu *Static Branch Prediction* dan *Dynamic Branch Prediction*

2.1.2.1 *Static Branch Prediction*

Static prediction adalah teknik *branch prediction* yang paling sederhana karena hanya menggunakan instruksi yang dimaksud untuk memperkirakan hasilnya dan tidak memasukkan *feedback* apapun dari *run-time environment*nya. Ini adalah kelebihan sekaligus kelemahan dari *static prediction*. Karena dengan tidak memperhatikan kelakuan *dynamic run-time* dari sebuah program, *branch prediction* tidak dapat beradaptasi dengan perubahan pada *pattern branch outcome*nya. *Pattern* ini dapat berbeda-beda berdasarkan dengan *input* set pada program atau fasa-fasa yang berbeda dari eksekusi program.

Kelebihan dari *static branch prediction* adalah kemudahannya untuk diimplementasikan, dan membutuhkan *resource hardware* yang sangat kecil. Algoritma *static branch prediction* sekarang ini sudah jarang dipakai, pemakaiannya sekarang ini lebih ke arah sebagai cara cadangan pada prosesor-prosesor yang menggunakan *dynamic prediction* apabila tidak ada informasi dari prosesor yang bisa dipakai oleh *dynamic predictor*.

Pada *single-direction static prediction*, *branch predictor* akan mengasumsi bahwa semua *branch instruction* yang datang adalah *taken* atau *not-taken*, sehingga instruksi berjalan dengan sangat cepat. Meskipun begitu, akurasi dari *static predictor* memiliki rata-rata 60-80%[1], dan usaha untuk memperbaiki kesalahan prediksi dari *static predictor* biasanya masih lebih cepat bila dibandingkan dengan menunggu proses *branch instruction* tanpa *predictor* (sehingga menimbulkan *bubble*).

Versi *static prediction* yang lebih kompleks akan mengambil *address* yang lebih rendah dari *address* saat ini, sehingga menambah akurasi prediksi *loop* dalam pemrograman.

Bentuk yang lebih kompleks dari *static prediction* mengasumsikan bahwa *backward branches*, yaitu *branch* yang memiliki *target address* yang nilainya lebih rendah dari dirinya, akan selalu diambil. Sementara *forward-pointing branches* tidak akan diambil. Teknik ini dapat membantu akurasi dari prediksi *loop*, yang biasanya terdiri dari backward-pointing *branch*es, dan lebih sering diambil daripada tidak.

2.1.2.2 *Dynamic Branch Prediction*

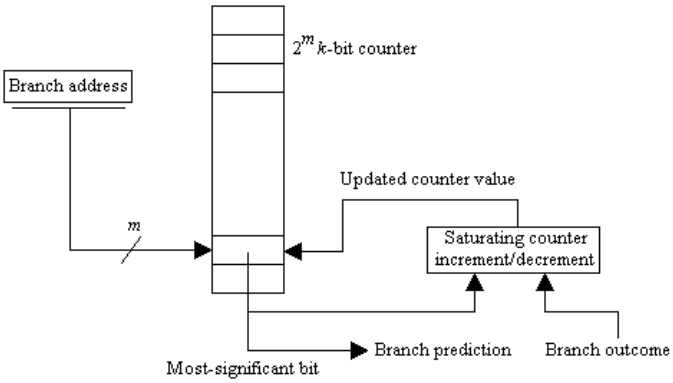
Walau *static branch prediction* dapat mencapai tingkat akurasi prediksi hingga sekitar 80%, bila informasi profilingnya tidak mewakili perilaku *run-time* yang sebenarnya, akurasi prediksi dapat menurun. Algoritma *dynamic branch prediction* mengambil keuntungan dari informasi *run-time* yang ada pada prosesor, dan dapat mengambil tindakan untuk *branch pattern* yang berubah-ubah. *Dynamic branch prediction* biasanya memiliki tingkat akurasi prediksi sekitar 80-95%[1].

Dynamic branch predictor dapat membutuhkan chipa area yang besar untuk diimplementasi, terutama bila algoritma yang kompleks dipakai. Untuk prosesor yang kecil seperti prosesor-prosesor generasi terdahulu, mungkin pemakaian *dynamic branch prediction* akan kurang efektif karena terlalu memakan tempat, tapi untuk prosesor-prosesor superscalar modern yang memiliki ukuran besar dan *pipeline* yang panjang, *dynamic branch prediction* dengan akurasi tinggi sangat dibutuhkan.

Teori dasar dari *dynamic branch predictor* adalah setiap kali prosesor menemui *outcome* yang sebenarnya dari *branch instruction* (*taken* atau *not-taken*), *predictor* akan mengingat semacam sebuah konteks agar bila *branch* yang sama ditemukan lagi, prediksi yang tepat dapat diambil.

Algoritma paling pertama dan paling sederhana yang dikemukakan untuk *dynamic branch predictor* adalah Smith's *algorithm*. *Predictor* ini terdiri dari sebuah tabel yang mencatat setiap hasil dari *branch*, apakah sebelumnya *taken* atau *not-taken*. Karena cara pelacakannya Smith *predictor* sering disebut sebagai *bimodal predictor*.

Smith *predictor* terdiri dari tabel 2^m *counter*, dimana masing-masing *counter* melacak arah *branch prediction* sebelum-sebelumnya. Karena tabel hanya memiliki 2^m *entry*, maka *branch address* (PC) perlu di*hash* menjadi *m*-bit. Fungsi *hashing* yang diajukan oleh Smith pada tahun 1981[7] adalah dengan memakai *low-order m*-bit dari *address*, atau dengan meng-XOR *low-order m*-bit dengan *m*-bit berikutnya yang lebih tinggi. Selanjutnya *index* dari *address* yang telah di-*hash* dipakai untuk meng-*address random* access *memory* yang berisikan keluaran dari *branch instruction* terakhir yang meng*index* lokasi yang sama. Kemudian diprediksikan hasil *branch*nya akan sama.

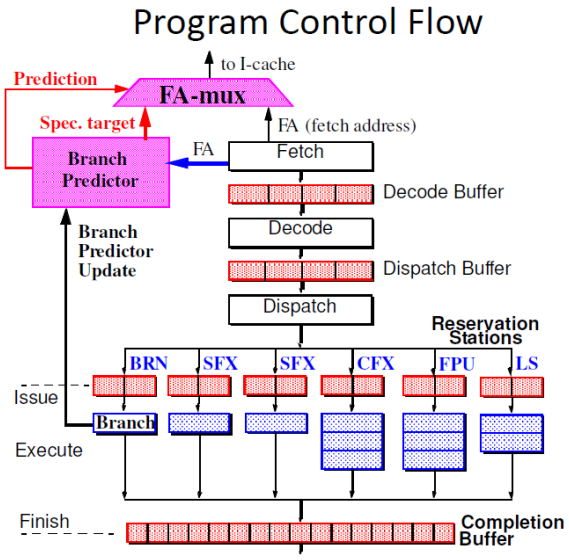


Gambar 2.6 Smith *predictor* dengan *entry* 2^m dari *saturating* k-bit *counter*[1]

Setiap *counter* pada tabel memiliki lebar *k*-bit, dan *most-significant* bit dari *counter* dipakai untuk arah *branch prediction*. Bila bitnya bernilai 1 maka *branch* dianggap sebagai *taken*, bila nilainya 0 dianggap *not-taken*.

2.1.3 Implementasi *Dynamic Branch Prediction* Pada Prosesor

Gambar dibawah adalah contoh skema dari rangkaian *branch predictor* pada program *control flow* prosesor.



Gambar 2.7 Mekanisme dari *branch predictor* pada program *control flow*[12]

Pada saat fasa *instruction fetch* (IF) dijalankan, prosesor mengirimkan isi dari PC ke dalam BTB. Kemudian BTB membandingkannya dengan *address* pada *cache*nya. Bila *address* terdapat dalam *cache*, berarti *branch* tersebut sudah pernah dieksekusi

sebelumnya. Sehingga kita dapat memprediksi bahwa *branch* tersebut akan memiliki kelakuan yang sama seperti sebelumnya, maka *predictor* akan mengirimkan *value* yang dimaksud ke PC.

Bila *address* tidak ada ditemukan, artinya bisa berarti kita belum pernah menemui *branch* tersebut sebelumnya (*compulsory miss)* atau kita pernah menemuinya tapi sudah lama sekali sehingga sudah digantikan dengan *branch* lainnya dalam *cache* BTB (*capacity miss* atau *conflict miss*). Ini menyebabkan kita tidak dapat melakukan prediksi pada *branch*, sehingga yang dapat dilakukan adalah memasukkan *input* PC+4, ini disebut juga dengan instruksi *predict not-taken*, karena kita tidak ingin mengambil *branch* tersebut dan ingin melanjutkan instruksi berikutnya dalam *memory*.

Tentunya prediksi bisa saja salah. Kita tidak dapat mengetahuinya sebelum masuk ke fasa *Instruction Decoding* (ID), tapi pada fasa IF kita tidak tahu instruksi jenis apa yang sedang dijalankan, oleh karena itu kita tetap melakukan prediksi. Selanjutnya kita membandingkan *branch target* yang telah dikalkulasi dengan *predicted* PC (dengan membandingkannya ke PC *register*. Bila prediksinya ternyata benar berarti tidak ada masalah, tapi bila prediksi salah (atau kita tidak melakukan prediksi karena tidak ditemukan dalam BTB), kita perlu men*stall pipeline* dan mengupdate BTB. Jadi secara ringkas, ada 3 situasi yang bisa terjadi dalam proses *branch prediction*.

1. *Branch* tidak ada dalam BTB
   Predictor mengupdate PC dengan PC+4. Saat *branch* sampai pada fasa ID, kita men*stall pipeline* untuk mengupdate BTB.

2. *Branch* ada dalam BTB tapi prediksi salah
   Predictor mengupdate PC dengan *predicted* PC. Saat *branch* sampai pada fasa ID, kita men*stall pipeline* untuk mengupdate BTB.

3. *Branch* ada dalam BTB dan prediksi benar
   Predictor mengupdate PC dengan *predicted* PC. Saat *branch* sampai pada fasa ID, tak ada yang perlu dilakukan

2.1.4 *Branch Misprediction*

Kesalahan prediksi pada *branch prediction* dapat terjadi karena berbagai macam hal. Beberapa *branch* memang susah untuk dipreksi, kemungkinan lainnya adalah karena pada kenyataannya *branch predictor* terbatas dalam segi ukuran dan kompleksitasnya.

Ada beberapa kasus dimana sebuah *branch* tidak dapat diprediksi. Salah satunya pada saat pertama kali *predictor* menemukan sebuah *branch*, dia tidak memiliki informasi sebelumnya tentang bagaimana *branch* tersebut berperilaku, sehingga hal terbaik yang dapat dilakukan *predictor* adalah mengambil pilihan *random* dan mengharapkan kemungkinan 50% benar. Dengan *predictor* yang memakai *branch history*, situasi yang sama terjadi setiap kali *predictor* menemui pola *branch history* yang baru. Prediktor perlu mempelajari *branch history* tersebut sebelum ia bisa memahami prediksi tepat untuk *branch* yag bersangkutan. Untuk *branch history* dengan panjang *n*, ada 2^n kemungkinan pola *branch history* yang terjadi, sehingga waktu *training* bertambah bersamaan dengan panjang *history*nya, kemungkinannya sangat kecil untuk *branch predictor* dapat memprediksi dengan baik pada masa pelatihan ini.

Kasus lainnya adalah bila data yang terlibat dalam program bersifat *random*. Sebagai contoh, bila sebuah program yang memproses data yang di*compress* dapat memiliki banyak *branch* yang susah untuk diprediksi karena *input* data yang di*compress* dengan baik akan bersifat *random*.

2.1.4.1 Penyebab *Miss* Pada PHT

Address-history dapat menjadi *miss* dalam PHT karena beberapa alasan berikut.

- ***Compulsory aliasing***

  Terjadi saat pertama kali pasangan *address-history* dipakai untuk meng*index* PHT. Satu-satunya cara untuk mengatasinya adalah untuk menginisialisasi PHT *counter* dengan cara tertentu sehingga sebagian besar dari *lookup* yang menghasilkan prediksi yang seakurat mungkin.

- ***Capacity aliasing***

  Terjadi karena ukuran dari working set dari pasangan *address-history* lebih besar dari kapasitas PHT. Ini dapat diatasi dengan menambah ukuran PHT.

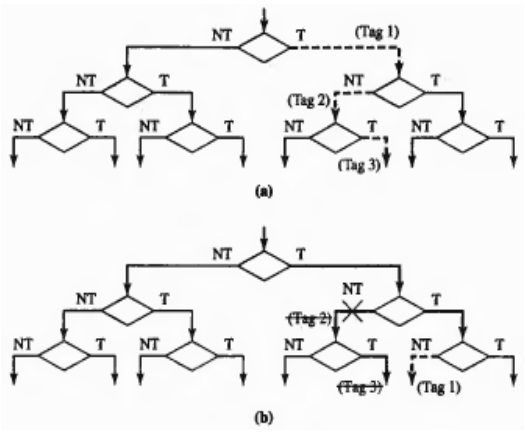
- ***Conflict aliasing***

  Terjadi saat 2 pasang *address-history* yang berbeda masuk ke dalam *entry* PHT yang sama. Menambahkan ukuran PHT biasanya tidak begitu memiliki pengaruh banyak untuk mengurangi *conflict aliasing*.

Salah satu cara untuk mereduksi kemungkinan konflik yang terjadi adalah dengan melakukan *hash*, yaitu dengan fungsi matematis yang mengkonversi data dalam jumlah besar menjadi *datum* kecil, pada *branch address* yang masuk ke dalam PHT agar tidak melebihi tabel *entry limit* yang ada.

2.1.4.2 *Branch Misprediction Recovery*

Branch prediction adalah teknik spekulasi yang membutuhkan mekanisme untuk menvalidasi spekulasi tersebut. *Dynamic branch prediction* dapat dianggap seperti memiliki dua mesin yang saling berinteraksi. Mesin pertama yang melakukan spekulasi, sementara mesin kedua melakukan validasi pada tahap-tahap akhir dalam *pipeline*. Apabila terjadi kesalahan prediksi, mesin yang kedua juga dapat melakukan koreksi. Ilustrasi dari kedua aspek dari *branch prediction* tadi dapat dilihat pada gambar dibawah.



Gambar 2.8 Dua aspek dari *branch prediction*: (a) *Branch Speculation* dan (b) *Branch* Validation/*Recovery*[1]

Gambar (a), yaitu aspek *branch Speculation*, mengilustrasikan spekulasi dari tiga *branch* dengan *branch* pertama dan ketiga diprediksi *taken* dan yang kedua diprediksi *not taken*. Ketiga spekulasi *basic block* ini diidentifikasi dengan *tag*. Instruksi yang di*tag* menandakan bahwa instruksi tersebut adalah spekulasi, dan nilai dari tiap *tag* mencerminkan *basic block* yang diwakilinya.

Gambar (b), yaitu aspek *branch validation/recovery*, mengilustrasikan proses validasi dari *branch* yang dispekulasi sebelumnya. Pada tahap ini, Apabila terjadi kesalahan prediksi, *predictor* mengupdate PC dengan *branch target* yang telah diperhitungkan (bila diprediksi *not-taken*) atau dengan *sequential instruction address* (bila diprediksi *taken*). Selanjutnya *predictor* menghilangkan *path* yang salah dengan men-*deallocate entry Re-order Buffer* (ROB) dan menginvalidasi seluruh instruksi pada *decode* dan *dispatch buffer*.

2.2 Simplescalar

Simulator prosesor adalah sebuah alat atau program yang dibuat untuk menghasilkan tiruan perilaku dari prosesor yang disimulasikan. Kinerjanya adalah mengolah *input* yang diberikan pemakai simulator dan menghasilkan *output* yang merupakan tiruan dari *output* yang dihasilkan oleh prosesor yang disimulasikan, hasil *output* ini dapat dipakai untuk melakukan revisi dalam prosesor tersebut. Maka dengan adanya simulator prosesor, dapat diuji kinerja dari sebuah prosesor sebelum prosesor itu dibuat, sehingga akan sangat menghemat biaya dan waktu.

Ada sekitar 40 simulator untuk melakukan simulasi dalam arsitektur komputer, seperti yang bisa dilihat dalam daftar simulator pada http://www.cs.wisc.edu/arch/www/tools.html. Dalam tugas akhir ini, penulis memakai simulator yang pertama dalam daftar tersebut, yaitu simplescalar. Simulator simplescalar dikembangkan oleh Todd Austin, dan hingga sekarang masih dipakai dalam kalangan akademi dan industri. Kelebihan-kelebihan dari simplescalar adalah:

- ***Highly flexible***

  Dapat mensimulasikan kinerja sekaligus performa dari prosesor

- ***Portable***

  Dapat dijalankan di hampir semua system yang menyerupai Unix, selain itu simulator simplescalar mendukung beberapa ISA (*Instruction* Set Arc*hit*ecture) sekaligus

- ***Extensible***

  Simplescalar memberikan *source code* yang lengkap untuk compiler, *library*, dan simulator.

- ***Performance***

  Simulasi prosesor yang dilakukan dengan simplescalar memakai *code* yang menghasilkan *output* hampir serupa dengan prosesor asli yang ingin dibuat.

Secara garis besar simulasi terbagi menjadi tiga kategori yaitu simulasi berdasarkan cakupan, detil, dan *input*. Tiap kategori tersebut terbagi lagi menjadi dua bagian simulasi , Berikut adalah penjelasan mengenai simulasi-simulasi tersebut:

a. Cakupan

- **Simulasi menyeluruh** atau penjadwalan instruksi adalah simulasi yang memasukkan banyak instruksi sekaligus dan melihat berapa banyak instruksi yang dapat diproses dengan patokan waktu tertentu.

- **Simulasi mikro-arsitektur** adalah simulasi yang melakukan pengecekan kondisi dari tiap bagian dalam prosesor per instruksi yang dimasukkan.

b. Detail

- **Simulasi fungsional** adalah simulasi yang melakukan pengujian pada prosesor dengan memasukkan suatu perintah yang mungkin dimasukkan oleh seorang programmer dan melihat apakah perintah tersebut dapat dieksekusi dengan benar.

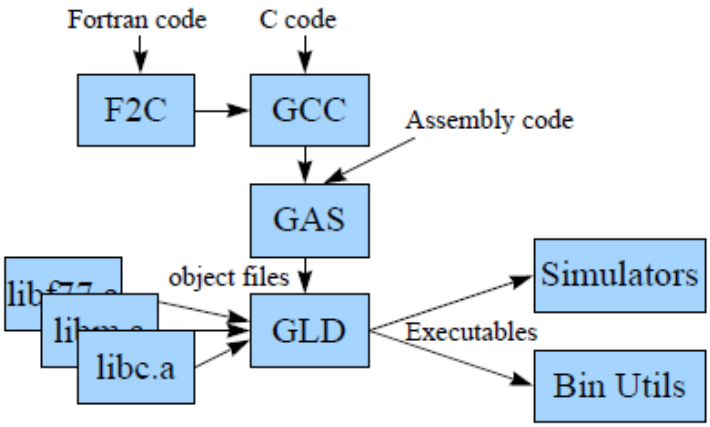
- **Simulasi performa** adalah simulasi yang hanya memperhatikan kecepatan dari sebuah eksekusi dari perintah yang dimasukkan.

c. *Input*

- **Simulasi *trace*** adalah simulasi yang melakukan tracing pada setiap instruksi yang dimasukkan.

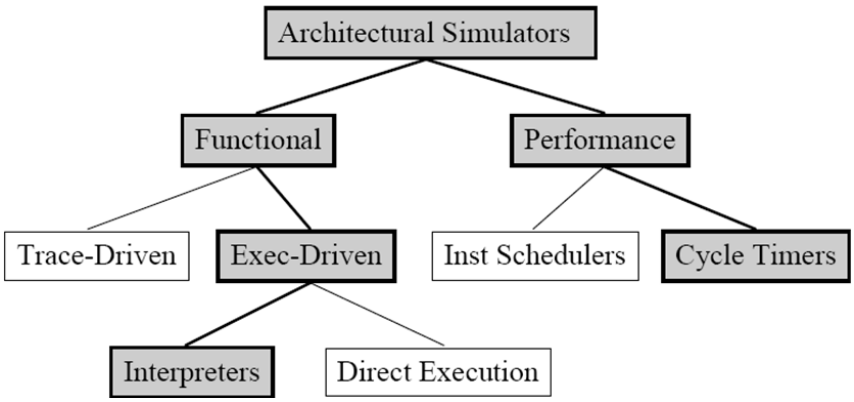
- **Simulasi eksekusi** adalah simulasi yang menjalankan instruksi-instruksi secara berurutan seperti sebuah program dan pada simulasi ini *register* dan memori pada prosesor tidak di *trace*.

Secara garis besar, cara kerja dari simulator simplescalar adalah sebagai berikut



Gambar 2.9 Diagram cara kerja dari simplescalar[13]

Salah satu hal yang membuat simplescalar dikatakan sebagai simulator yang fleksibel adalah karena simplescalar memiliki paket-paket yang dapat melakukan beberapa jenis simulasi tersebut. Berikut adalah bagan klasifikasi dari simulator-simulator (bagian-bagian yang berwarna keabuan adalah bagian-bagian yang dapat terdapat pada simplescalar *tool set*).



Gambar 2.10 Klasifikasi dari simulator (bagian abu-abu adalah bagian yang ada pada simplescalar)[12]

Simplescalar memiliki 6 buah paket simulasi, yaitu sim-fast, sim-safe, sim-profile, sim-cache, sim-bpred, dan sim-outorder.

- Sim-fast

  Paket simulasi penerjemah instruksi yang cepat, difokuskan dalam kecepatannya. Simulator ini tidak memperhitungkan behavior dari *pipeline*, *cache*, atau bagian lain dari microarc*hit*ecture

- Sim-safe

  Paket simulasi penerjemah instruksi yang lebih lambat dari sim-fast, karena paket simulasi ini memeriksa *memory alignment* dan *memory* access *permission* dari semua operasi *memory*. Simulator ini dapat dipakai bila program yang disimulasikan menyebabkan paket simulasi sim-fast menjadi crash tanpa kejelasan.

- Sim-profile

  Paket simulasi penerjemah dan profiler dari instruksi. Simulator ini mencatat dan melaporkan jumlah *dynamic instruction* dan *instruction* class, pemakaian *address* mode, dan profile dari text dan segmen data.

- Sim-cache

Paket simulasi *memory* sistem. Simulator ini dapat meniru sebuah sistem dengan sejumlah level instruksi dan data *cache*, dimana masing-masing dapat dikonfigurasi untuk ukuran-ukuran dan organisasi yang berbeda. Simulator ini ideal untuk simulasi *fast cache*, apabila efek dari *cache* performance pada waktu eksekusi tidak diperlukan

- Sim-bpred

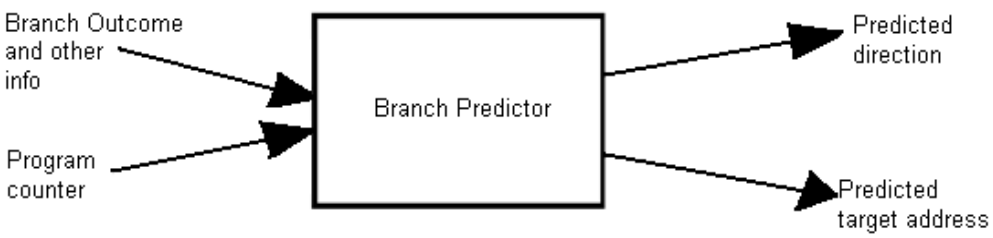
  Paket simulasi *branch predictor*. Paket ini dapat mensimulasikan skema *branch prediction* yang berbeda-beda dan melaporkan hasilnya, seperti ketepatan dan ketidak-tepatan prediksinya. Seperti sim-*cache*, paket ini tidak dapat mensimulasikan efek *branch prediction* pada waktu eksekusi secara akurat

- Sim-outorder

  Paket simulasi *microarchitectural* yang mendetail. Paket ini memodelkan mikroprosesor secara detail, termasuk *branch prediction*, *cache*, dan *memory* eksternal. Simulator ini memiliki parameter tinggi dan dapat meniru mesin-mesin yang memiliki jumlah unit eksekusi yang berbeda-beda

2.2.1 Sim-bpred

Paket simulasi yang dipakai oleh penulis dalam tugas akhir ini adalah paket simulasi sim-bpred dengan tujuan untuk melakukan *branch prediction*. Sebuah simulator *branch prediction* dapat digambarkan sebagai berikut



Gambar 2.11 Gambaran simulator *branch prediction*

Pada sistem *branch prediction*, mikroprosesor akan mencoba untuk memperkirakan apakah *branch instruction* akan berpindah atau tidak, dengan berdasarkan memori dari *branch-branch* sebelumnya. Sebagai contoh, apabila sebelumnya sudah berpindah pada 4 instruksi sebelumnya, ada kemungkinan pada instruksi berikutnya juga akan berpindah lagi.

Dengan paket simulasi sim-bpred, kita dapat mengolah informasi *instruction* pointer dari *input* dan memperoleh informasi dimana instruksi berikutnya berada dalam *instruction* stream serta arah dari *branch* dan *target* dari *branch* (tujuan dari *branch*) tersebut pada *output*. *Target* dari *branch* perlu diperhitungkan lagi, karena bila sebuah *branch* sudah diperhitungkan, *address*nya akan berubah.

Apabila terjadi kesalahan dalam prediksi, maka *predictor* perlu melakukan *recovery* untuk membetulkan kesalahan dalam prediksinya dan mengurangi kemungkinan kesalahan yang sama terjadi.

2.2.1.1 Cara Kerja Dari sim-bpred

Ada beberapa langkah-langkah kerja yang dilakukan oleh sim-bpred. Yang pertama setelah datangnya *input* dari user, sim-bpred memanggil *function* `sim_reg_options` untuk melakukan *register* tipe simulator yang dimasukkan oleh user, selanjutnya sim-bpred memanggil *function* `sim_check_options` untuk memeriksa tipe simulator yang dimasukkan.

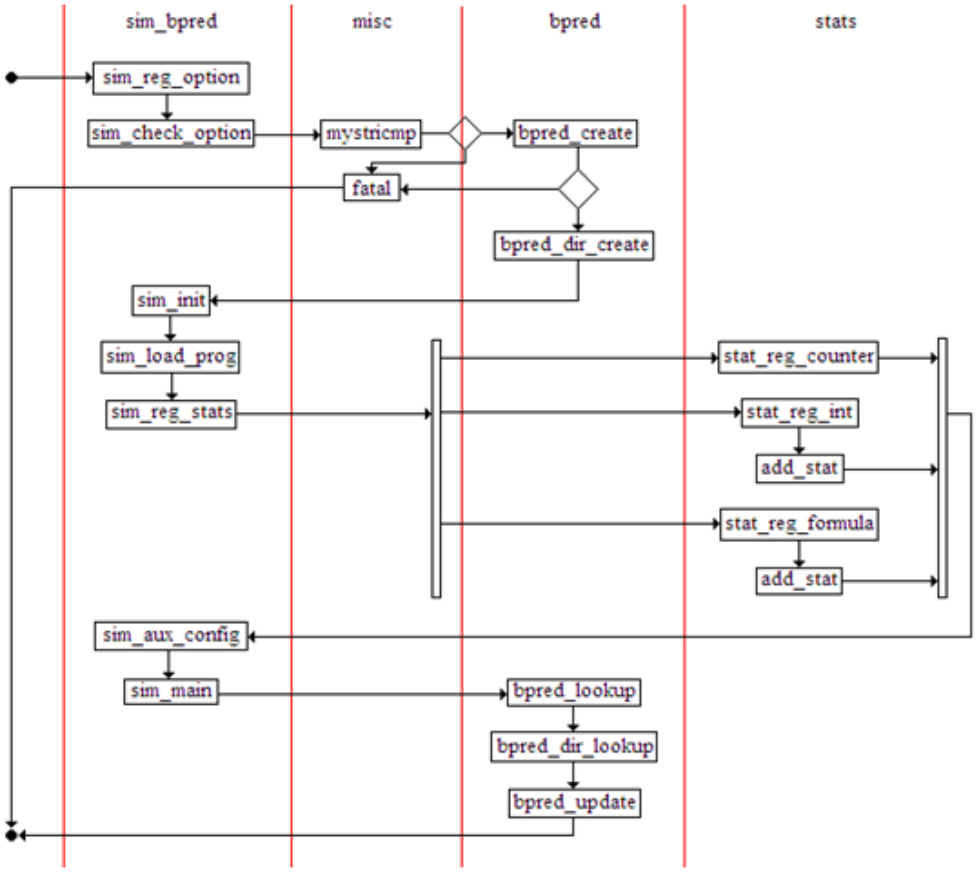
Proses selanjutnya adalah melakukan pembandingan string dengan memanggil *function* mystricmp dan memanggil *function* `bpred_create` untuk mengolah *branch predictor* yang dimasukkan. Apabila pada salah satu dari kedua *function* di atas terjadi kesalahan *input*, seperti misalkan terjadi kesalahan pengetikan atau *branch predictor* yang dimasukkan tidak terdapat dalam `sim_bpred`, maka proses dihentikan dan kembali ke langkah awal.

Kemudian simulator yang dimasukkan diinisialisasi dengan memanggil *function* `sim_init`, lalu program diload ke *state* simulated dengan `sim_load_prog`, dan dilakukan *register* statistik simulator-specific dengan pemanggilan *function*

`sim_reg_stats` (yang di dalamnya dilakukan pemanggilan *function-function* `stat_reg_counter`, `stat_reg_int`, dan `stat_reg_formula`).

Langkah selanjutnya adalah mencetak informasi konfigurasi *auxiliary* simulator-specific dengan *function* `sim_aux_config`, setelah itu barulah fungsi utama dari sim-bpred dijalankan dengan memanggil *function* `sim_main`.

Pemanggilan-pemanggilan *function branch prediction* yang dilakukan oleh sim-bpred adalah pemanggilan *function* `bpred_lookup` untuk mendapatkan *predicted fetch address* berikutnya, kemudian dilakukan pemanggilan *function* `bpred_dir_lookup` untuk memprediksi arah *branch* dan melakukan update dari hasil *branch instruction* dengan memanggil *function* `bpred_update`. Ilustrasi dari simulasi dapat dilihat dari ilustrasi gambar berikut



Gambar 2.12 Ilustrasi langkah-langkah kerja sim-bpred

2.2.1.2 Beberapa Fungsi Dasar sim-bpred

Branch prediction sendiri memiliki beberapa arsitektur dasar, diantaranya adalah berikut ini

2.2.1.2.1 *Two-level Adaptive Predictor*

Kadang kala *branch instruction* berhubungan satu sama lain, dan informasi statistik dari *branch* execution *history* tidak dapat memprediksi hasil dari *branch* dengan baik. Maka para peneliti memakai multi-level *predictor*. Dalam simplescalar, diimplementasikan fungsi *two-level predictor*.

Two-level predictor melakukan analisa instruksi-instruksi sebelumnya dan menyimpan pola yang ditemukannya dalam tabel *register*. Metode ini mengingat *history* dari kejadian *n* terakhir dari *branch* dan memakai satu *saturating counter* untuk setiap 2^n *history pattern*.

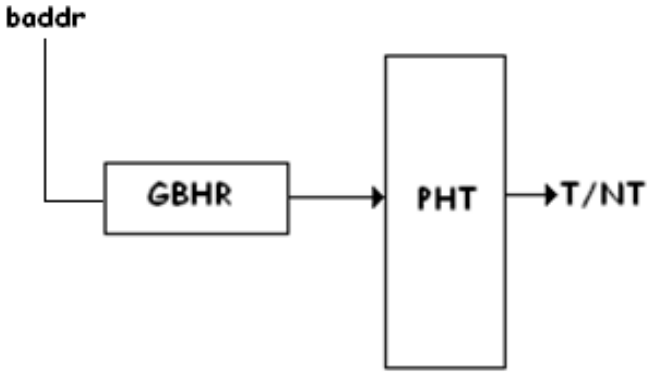
Sebagai contoh untuk $n = 2$. Ini menunjukkan bahwa dua kejadian terakhir pada *branch* disimpan pada 2-bit shift *register*. *Register* ini bisa memiliki 4 macam nilai biner yang berbeda: 00, 01, 10, 11. Dimana 0 berarti "not *taken*" dan 1 berarti "*taken*". Sekarang kita membuat sebuat tabel yang dinamakan *Pattern History Table* (PHT) dengan empat *entry*, masing-masing untuk setiap $2^n = 4$ *branch history* yang memungkinkan. Setiap *entry* dalam PHT mewakili 2-bit *saturating counter* seperti pada gambar 2.4. *Branch history register* dipakai untuk memilih yang mana dari empat *saturating counter* yang akan dipakai. Bila *history*nya bernilai 00, maka *counter* pertama yang dipakai, bila bernilai 11, maka *counter* terakhir yang dipakai.

Ada dua cara untuk mengimplementasikan BHR, yaitu dengan cara *global* (G) atau dengan cara *individual*/*local* (P). *Global* BHR dishare dengan semua *static branch*, sementara *local* BHR dishare kepada setiap setiap *static branch* atau subset dari *static branch*.

Ada tiga cara untuk mengimplementasikan PHT, yaitu dengan cara *global* (g), individual (p), atau shared (s). *Global* PHT memakai sebuah tabel untuk mendukung

prediksi dari semua *static branch*, sementara individual PHT dapat dipakai saat masing-masing PHT dibagikan ke setiap *static branch* (p) atau subset *static branch* (s). Saat *history*-based FSM dipakai untuk mengimplementasikan prediction *algorithm*, Yeh dan Patt[10] menerjemahkannya sebagai *adaptive* (A).

Variasi pertama dari *two-level predictor* adalah *global-history two-level predictor*, dimana *predictor* memakai *history* hasil dari sejumlah tertentu instruksi-instruksi *branch* yang baru dieksekusi. Hasil-hasil ini disimpan dalam *register* yang disebut *Global Branch History Register* (GBHR). GBHR dengan *n*-bit akan merekam kelakuan *n* instruksi *branch*. GBHR ini yang kemudian menjadi indeks untuk PHT, sehingga predisksi dilakukan berdasarkan kelakuan instruksi *branch* secara *global*. Dengan *h* bit *branch history* dan *m* bit *branch address*, PHT memiliki 2^{h+m} *entry*. Saat memakai *m*-bit *branch address* (dimana *m* lebih kecil dari total kelebaran dari PC), *branch address* harus di *hash* menjadi *m*-bit, seperti dengan Smith *predictor*. Dari cara implementasinya yang memakai *adaptive predictor* (A) dengan *global* BHR (G) dan PHT single table yang mendukung prediksi *static branch* (g), *Global-history predictor* juga disebut dengan *GAg predictor*.



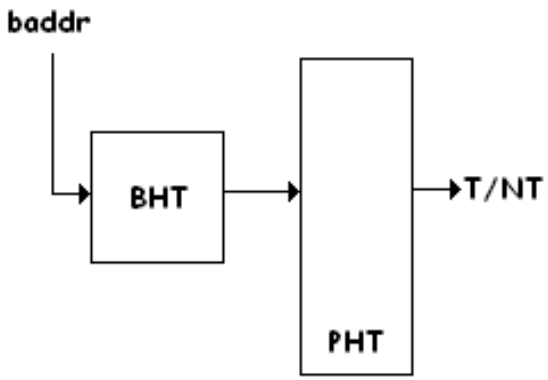
Gambar 2.13 Gambaran dari *Global-history predictor*

Variasi kedua adalah *local-history two-level predictor*, dimana *predictor* melacak dan menyimpan pola dari kelakuan instruksi *branch* tertentu saja. Untuk implementasinya, *local predictor* memakai BHR secara individual untuk setiap *branch* (bentuk ini disebut juga *per-branch*), kumpulan BHR ini membentuk sebuah *Branch History Table* (BHT). Setiap baris BHT yang terdiri dari *n*-bit akan mampu menyimpan *n* kelakuan terakhir dari instruksi *branch* tertentu. BHT ini yang

kemudian digunakan sebagai indeks PHT, sehingga prediksi dilakukan berdasarkan kelakuan instruksi *branch* secara *local*.

Local-history two-level predictor yang memakai *L*-bit *entry* BHT dan *h*-bit *history* dan memakai *m*-bit *branch address* untuk *index* PHT, membutuhkan ukuran total *Lh* + 2^{h+m+1} bit. *Lh* bit adalah untuk BHT, dan PHT memiliki 2^{h+m} *entry*, yang masing-masing lebarnya 2-bit (+1 pada exponennya).

Dari cara implementasinya yang memakai *adaptive predictor* (A) dengan individual BHR (P) dan PHT single table yang mendukung prediksi *static branch* (g), *Local-history predictor* juga disebut dengan *PAg predictor*.

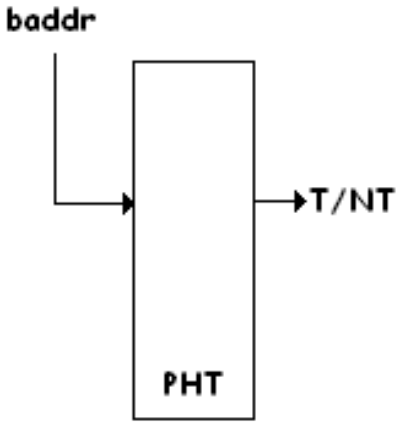


Gambar 2.14 Gambaran dari *Local-history predictor*

Sebagai contoh, kita ambil sebuah *Condition*al jump yang diambil setiap 3 kali *branch* dengan *branch* sequence 001001001. Untuk kasus ini, *entry* 00 pada PHT akan masuk ke *state* "TT" (*Strongly taken*), mengindikasikan setelah dua kali nilai 0 akan muncul nilai 1. *Entry* 01 akan masuk ke *state* "NN" (*Strongly not-taken*), mengindikasikan setelah 01 akan muncul nilai 0.

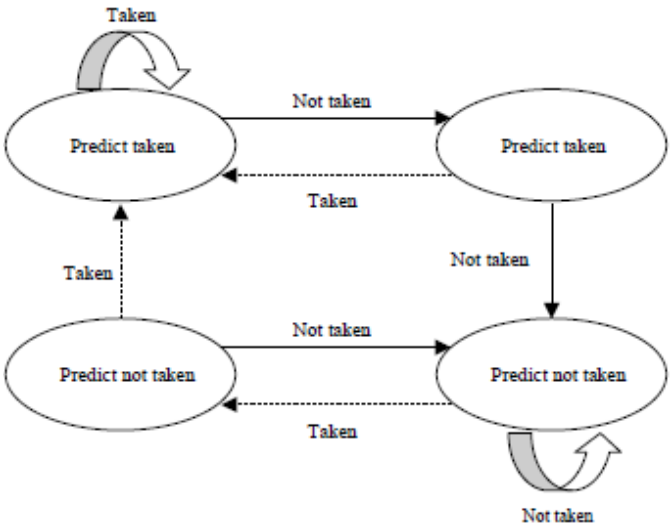
2.2.1.2.2 *Bimodal Predictor*

Bimodal predictor adalah satu *branch predictor* yang paling sederhana yang melacak perilaku setiap *branch* secara individu. *Bimodal branch predictor* mempunyai kelebihan karena disini *branch* dapat dianggap *taken* atau *not taken*. Distribusi *bimodal* dari perilaku *branch* memungkinkan desainer untuk mewakili kejadian suatu *branch* dengan satu bit.



Gambar 2.15 Gambaran dari *Bimodal predictor*

Untuk setiap *branch* yang diambil, *counter* yang sesuai akan bertambah, sedangkan untuk yang tidak diambil, *counter* yang sesuai akan berkurang. Bit yang paling signifikan dari *counter* digunakan untuk prediksi, 1 berarti *taken*, 0 berarti *not-taken*. Dengan cara ini, *branch* yang berulang kali *taken* akan diprediksi secara akurat, begitu juga *branch* yang berulang kali *not-taken*. Sifat dari 2-bit *saturating counter* didefinisikan melalui *state* machine yang ditunjukkan oleh gambar dibawah.

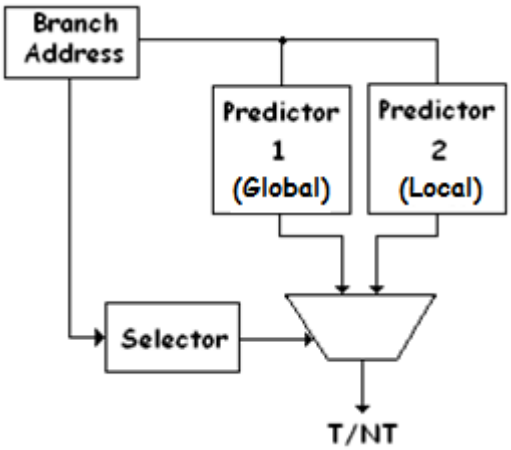


Gambar 2.16 Gambaran 2-bit *counter* pada *bimodal predictor*[5]

Kelebihan dari 2-bit *counter* dibandingkan dengan 1-bit adalah *Conditiona*l jump harus menyimpang dua kali dari biasanya sebelum perubahan prediksi, sehingga kesalahan prediksi menjadi lebih kecil.

2.2.1.2.3 *Hybrid Predictor*

Hybrid predictor, atau juga disebut *combined predictor*, menerapkan lebih dari satu mekanisme prediksi. Konsep dasarnya adalah menggabungkan *global predictor* dengan *local predictor* dengan maksud menggabungkan kemampuan kedua *branch prediction* ini. Konsep *Hybrid predictor* ini dikemukakan oleh Scott McFarling.



Gambar 2.17 Gambaran dari combined/*hybrid predictor*

Konsep *Hybrid predictor* dibuat karena beberapa *branch* akan dapat diprediksi dengan lebih akurat dengan memakai *global history*, sementara yang lainnya akan lebih baik bila menggunakan *local history*. Sementara program biasanya memiliki campuran dari tipe-tipe *branch* tersebut. *Hybrid predictor* memonitor jenis *history predictor* mana yang memiliki performa lebih baik untuk suatu *branch*, dan menggunakan salah satu dari mekanisme seleksi untuk memilih di antara mereka.

Scott McFarling mengusulkan menggunakan struktur *bimodal* sebagai *selector* dengan array 2-bit *counter* diperbaharui dengan akurasi prediksi dari dua prediktor digunakan sebagai pengganti apakah cabang diambil atau tidak-diambil.

2.2.2 *Perceptron Predictor*

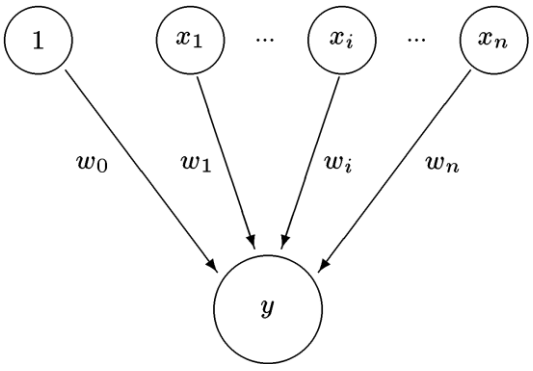
Perceptron adalah properti dari *neural network* yang memungkinkan pembelajaran dan identifikasi dari koneksi antar objek yang lebih kuat dan yang lebih lemah. Setelah jumlah pembelajaran tertentu, pada ketidakpastian, atau setelah mengalami kesalahan, dia kembali mengajar diri sendiri

Pemakaian dasar dari *Perceptron* adalah untuk meng-*assign* vektor *weight* untuk setiap objek dalam sistem dimana masing-masing *weight* mewakili kekuatan hubungan antara suatu *perceptron* pada objek dengan objek lainnya. Semakin kuat hubungan, semakin besar *weight* yang mewakilinya.

Perceptron predictor adalah bentuk paling sederhana dari *neural predictor*. Pada *perceptron predictor*, masing-masing *branch address* dipetakan menjadi satu entri pada tabel *perceptron*. Setiap entri pada tabel ini terdiri dari *state perceptron* tunggal.

Keuntungan utama dari *perceptron predictor* adalah kemampuannya untuk mengeksploitasi *history* yang panjang sementara hanya membutuhkan pertumbuhan *resource* linear. *Predictor* biasa membutuhkan pertumbuhan *resource* eksponensial. Jimenez melaporkan peningkatan *global* sebesar 5,7% lebih dari *hybrid prediction* McFarling[8].

Namun kerugian utama dari *perceptron predictor* adalah *latency* tinggi. Bahkan setelah mengambil keuntungan dari trik aritmatika kecepatan tinggi, latency perhitungannya relatif lebih tinggi dibandingkan dengan periode clock dari mikroarsitektur modern. Untuk mengurangi latency prediksi, pada tahun 2003 Jimenez mengusulkan *fast-path neural predictor*, di mana *perceptron predictor* memilih beban yang dipakainya sesuai dengan jalur *branch* yang dipakai, dan bukan menurut *branch* dari PC.



Gambar 2.18 Gambaran dari algoritma *perceptron*[8]

Perceptron predictor memiliki algoritma yang dipakai untuk pembelajaran secara otomatis agar mencapai hasil prediksi yang optimal. Pada *perceptron branch*

predictor, setiap bit bit x_i dari *input x* akan bernilai '1' bila *branch*nya *taken* (BHR_i = 1) dan x_i bernilai '-1' bila *branch*nya *not-taken* (BHR_i = 0), namun ada satu bias *input* x_o yang nilainya selalu 1. *Perceptron* memiliki satu nilai *weight* w_i untuk setiap *input* x_i, termasuk juga *weight* w_o untuk *input* biasnya (x_o). Perhitungan *output y* dari *perceptron* adalah sebagai berikut.

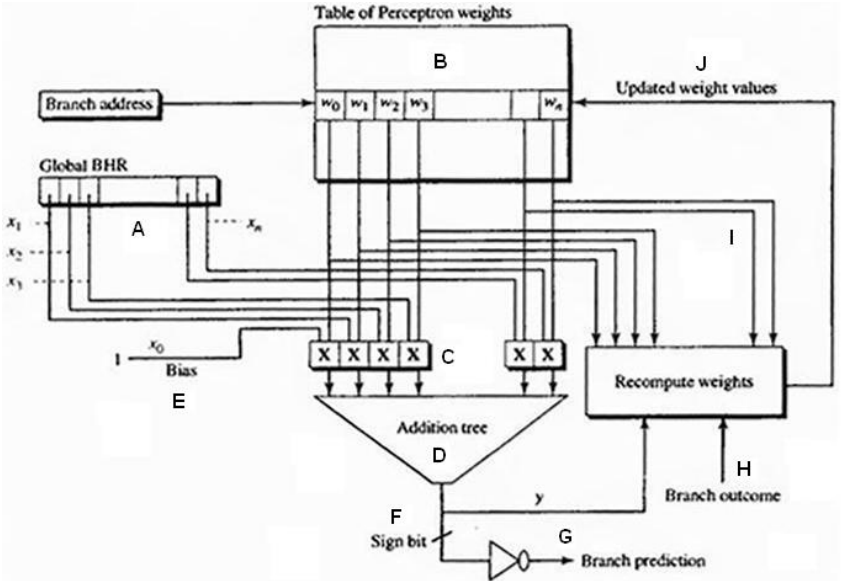
$$y = w_0 + \sum_{i=1}^{n} x_i w_i$$

Dot product dari *input* vector dan *weight* vector adalah hasil *output y* pada network. Bila *y* menghasilkan nilai negatif, maka *branch* diprediksi sebagai *not-taken*, dan sebaliknya bila *y* bernilai positif.

Setelah *branch outcome* sudah terkirim, *weight* dari *perceptron* dapat diupdate. Kita ambil nilai dari *t* = -1 bila *branch not-taken* dan *t* = 1 bila *branch taken*. Dan kita ambil nilai *training threshold* sebagai $\theta > 0$. Variabel y_{out} dilihat sebagai berikut

$$y_{out} = \begin{cases} 1 & \text{if } y > \theta \\ 0 & \text{if } -\theta \le y \le \theta \\ -1 & \text{if } y < -\theta \end{cases}$$

Bila nilai y_{out} tidak sama dengan t, seluruh *weight* diupdate dengan rumus $w_i = w_i + t\, x_i$, $I \in \{0, 1, 2, \ldots, n\}$. Sementara itu, nilai $-\theta \le y \le \theta$ menandakan *perceptron* belum terlatih ke tahap dimana prediksi dapat diambil dengan kemungkinan tinggi. Dengan mengatur y_{out} menjadi 0, kondisi $y_{out} \neq t$ akan selalu bersifat true dan *weight* pada *perceptron* akan diupdate (proses *training* berlanjut).



Gambar 2.19 Gambaran diagram *hardware organization* dari *perceptron predictor*[1]

Gambar 2.19 diatas adalah diagram *hardware organization* dari *perceptron predictor*. Bagian (A) adalah kolom isi dari BHR, yaitu *input* x_i pada *branch predictor*. Setiap *input* x_i diassign *weight* w_i dari tabel *perceptron weight* (B), yang berisikan *branch address* yang sudah di *hash* lalu nilai keduanya dikalikan di dalam X (C). Kemudian seluruh hasilnya kemudian dijumlahkan pada *Addition Tree* (D).

Input bias x_o yang selalu bernilai 1 (E) juga dikalikan dengan sebuah *weight* w_o dari tabel *perceptron weight* dan dimasukkan ke dalam penjumlahan pada *Addition Tree* (D).

Hasil prediksinya dibuat berdasarkan *most-significant* bit dari *y*. Sesuai dengan rumus sebelumnya, bila hasil penjumlahannya < -θ, maka $y = -1$, dan sebaliknya jika hasil penjumlahannya > θ, maka $y = 1$. Sign bit pada rangkaian (F) bertugas untuk mengecek apakah nilai dari *y* positif atau negatif. Bila hasilnya adalah positif, maka *branch* akan diprediksi sebagai *taken*, tapi bila negatif maka *branch* akan diprediksi sebagai *not-taken* (G).

Setelah melakukan prediksi kemudian diproses, nilai *y* dibandingkan dengan actual *Branch Outcome* (H), yaitu hasil *branch* sebenarnya dari tahap ID. Nilai dari *branch outcome* juga berbentuk '1' dan '-1' untuk hasil *taken* dan *not-taken*.

Kemudian rangkaian melakukan penghitungan ulang *weight* pada kolom *Recompute Weight* (I). Pada tahap ini, perlu diperhatikan dua hal berikut:

- jika *branch outcome* = nilai *y*, maka tidak perlu dilakukan perhitungan ulang, dan semua *weight* bernilai tetap
- jika tidak, maka dicari selisih *weight* baru dengan rumus $w_i = w_i + t\, x_i$, $I \in \{0, 1, 2, \ldots, n\}$. Kemudian hasilnya dijumlahkan dengan *weight* awal (I) untuk mendapatkan *weight* baru (J).

Perceptron predictor memiliki 4 parameter; jumlah *perceptron*, jumlah bit yang dipakai oleh *history*, lebar dari *weight*, serta learning *threshold*. Ada sebuah standar untuk nilai optimal *threshold* sebagai *function* dari *history* length (*h*) yang dilakukan oleh para peneliti terdahulu[8], dimana *threshold* θ nilainya sama dengan $[1.93h + 14]$. Ini dikarekan penambahan *weight* lain ke dalam *perceptron* akan meningkatkan *output* rata-ratanya sebesar sebuah konstanta tertentu, maka *threshold*nya juga harus ditambahkan sebesar sebuah konstanta juga, sehingga menghasilkan hubungan linear antara panjang *history* (*h)* dengan *threshold* (θ).