



به نام خدا

دانشگاه صنعتی شریف – دانشکده مهندسی برق

استاد درس: دکتر حاج صادقی

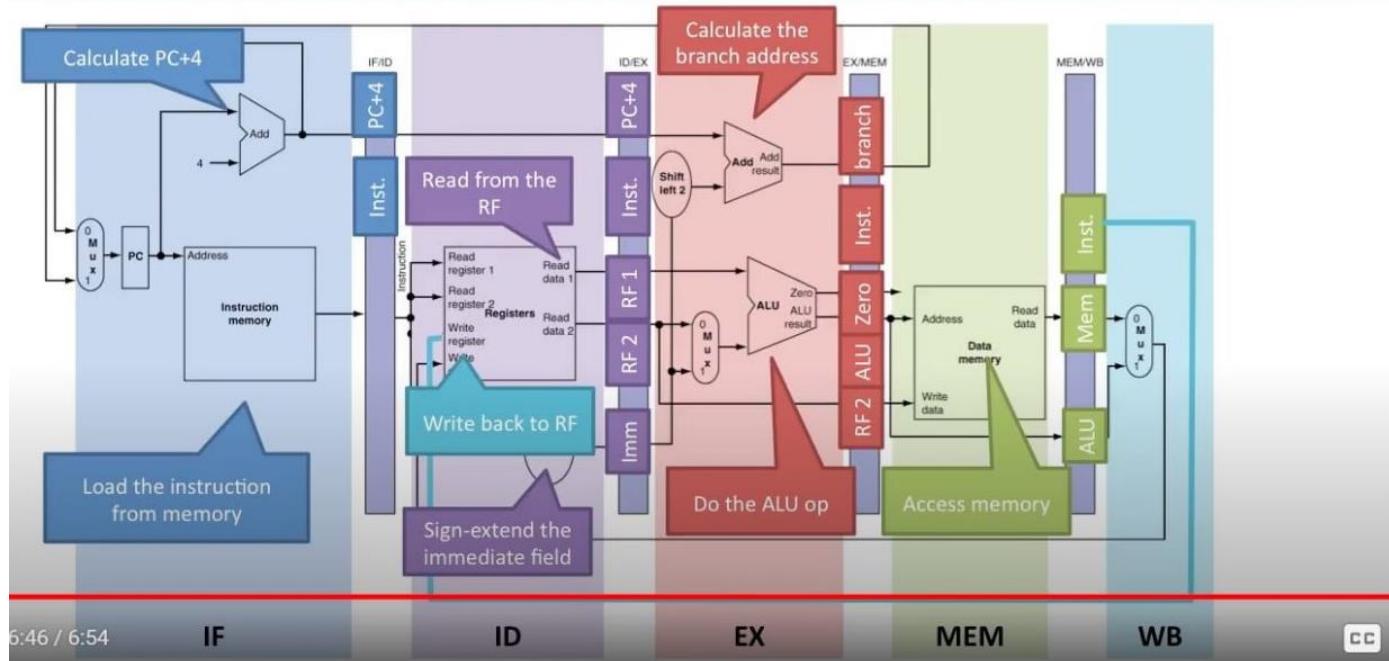
زهرا مجتبیدین ۹۹۱۰۲۱۶۷ – سارا رضانژاد ۱۶۴۳

هدف کلی پروژه درس

هدف این پروژه طراحی پردازنده میپس پایپ لاین است که دارای پنج مرحله است و بین هر دو مرحله یک بافر قرار دارد.

The MIPS pipeline

51



○ توضیحات به شرح زیر می‌باشند:

- در مرحله اول اینستراکشن *Fetch* میشود و در اختیار مراحل بعدی قرار میگیرد. در این مرحله یک مالتی پلکسر موجود است که می توان به وسیله‌ی آن به ورودی *Program Counter* مقدار مناسب را داد.
- سپس در مرحله دیکد اینستراکشن را پارس کرده و مقادیر مختلف را از جمله نوع *operand* دو رجیستر مبدا و رجیستر مقصد و همچنین مقدار *Immediate Offset* یا *Immediate* را به دست می آوریم و به همراه رجیسترها در

اختیار مرحله بعدی که *Execution* هست قرار میدهیم.

۳. در این مرحله آدرس موثر *Branch* را محاسبه کرده و همچنین علمیات ریاضی مناسب را انجام خواهیم داد. لازم به ذکر است قبل از دو ورودی واحد محاسبه گر یک مالتی پلکسسور چهار ورودی قرار دارد. که توسط واحد کنترل *Forwarding* میشود.

۴. در مرحله‌ی بعدی از *Memory* می‌خوانیم یا در آن مینویسیم و مقدار خوانده شده را به واحد آخر یعنی *Right Back* انتقال میدهیم.

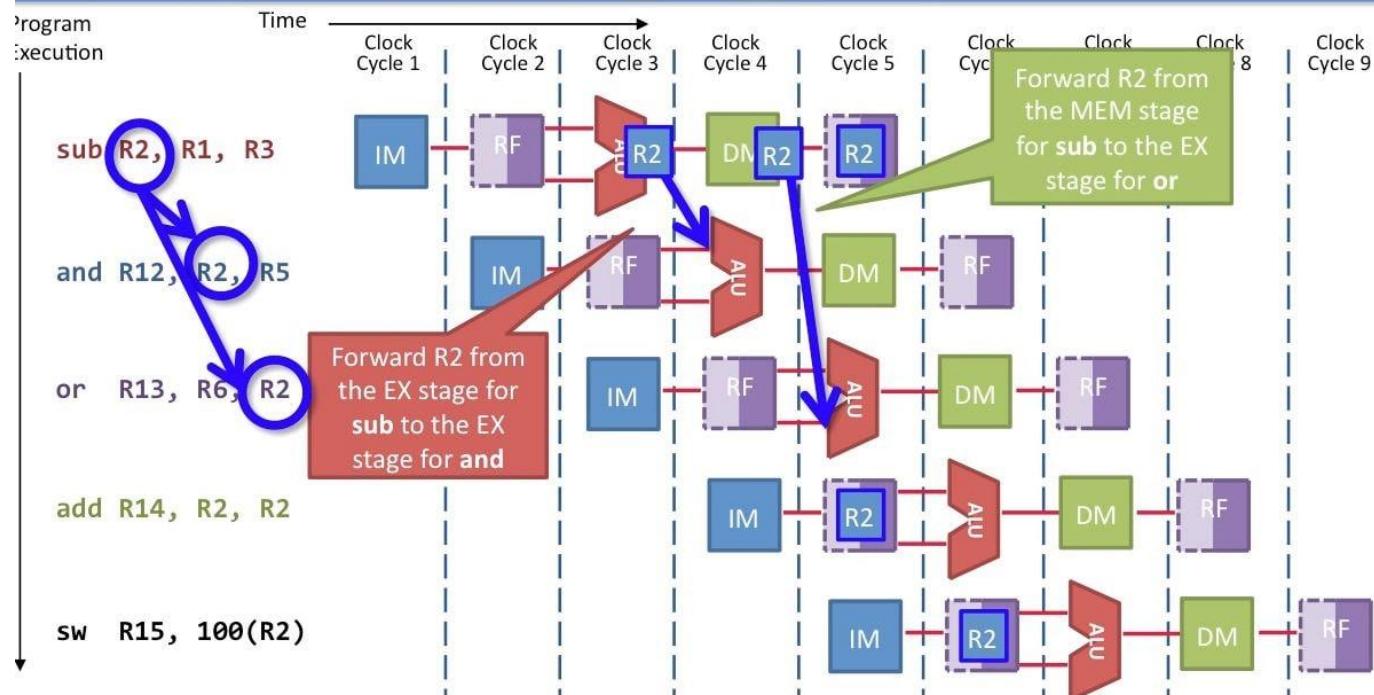
۵. در مرحله‌ی آخر در صورت آنکه مقدار محاسبه شده باید در رجیستر مقصد نوشته شود این کار را انجام میدهیم، که میتواند خروجی واحد محاسبه گر یا *Memory* باشد.

۶. چند واحد دیگر نیز خواهیم داشت.

۷. از جمله *Control Forwarding* و *Taking a guess* گر پرش.

۸. در *Controller* در صورت وقوع پرش مقدار مناسب را برای *Select Program Counter* انتخاب میکنیم. در واحد *Forwarding* در صورت وابستگی اینستراکشن در حال اجرا به رجیستری که هنوز مقدار مناسب در آن نوشته نشده یعنی دستور قبلی تمام نشده با فوروارد کردن *Data* از مراحل *Memory* و *Data* به هر کدام از دو ورودی واحد محاسبه گر وابستگی و ها ارد را رفع میکند.

Where does forwarding help?



وابستگی ها از جمله دو عملیات ریاضی پشت سر هم که رجیستر ورودی دستور اخر از رجیستر خروجی دستور قبل یا دو دستور قبل استفاده کند. بعد از *Load* یا *Store* یا عملیات ریاضی؛ اگر بعد از لود از عملیات ریاضی یا *Branch* یا *conflict* استفاده کنیم به نحوی که خروجی لود ورودی این اینستراکشن ها باشند. دستور بعد از *Store* نخواهد داشت. بعد از *Right Back Branch* چون *Nadarim* وابستگی نخواهیم داشت.

نوع *Branch Predictor* مورد نظر:

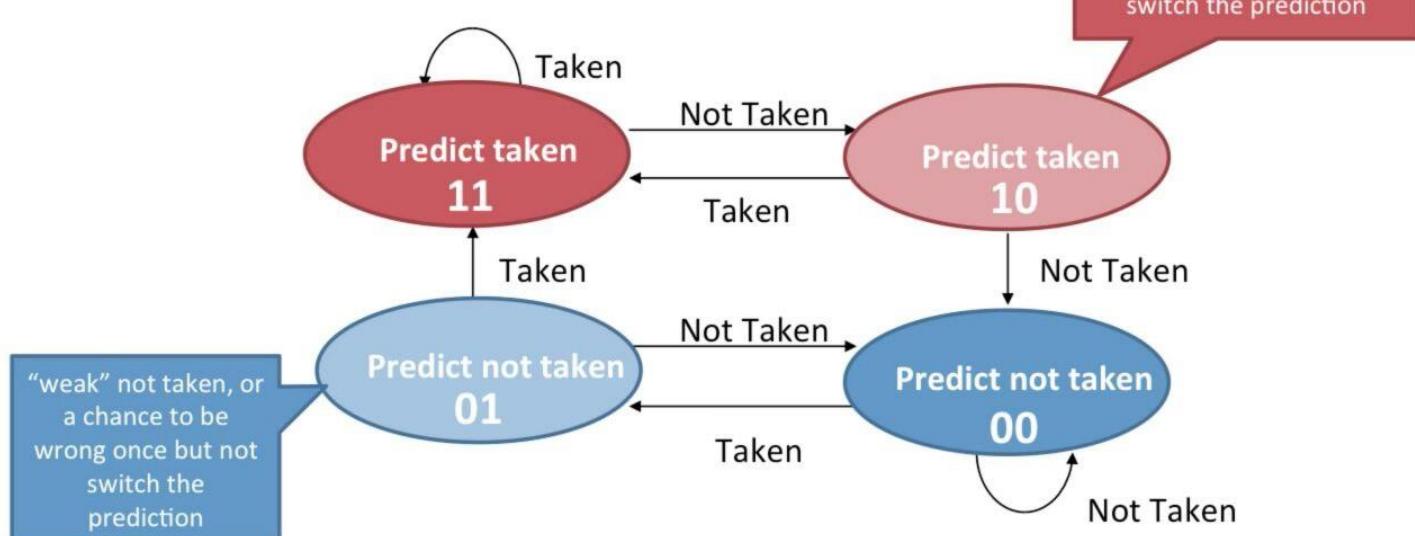
Algorithm	Number of misses
Static: Not Taken	a
Dynamic: Last time predictor	b
Dynamic: Two-bit Counter based prediction	c
Dynamic: Global branch prediction	d
Dynamic: Gshare branch prediction	e
Dynamic: Local branch prediction	f

در استفاده از یک ماشین حالت و بررسی دو ورودی قبلی حدس میزنیم.

2-bit branch predictor

32

- If 1-bit isn't good enough, how about 2-bits?
 - More "detailed" branch behavior



یک استیت ماشینی می باشد که استیت اولیه آن صفر می باشد. اگر در جایگاه صفر، صفر باید همانجا مانده و اگر که یک باید به استیت 1 می رویم. یا مثلًا اگر در استیت 3 باشیم و صفر باید به استیت 2 می رویم.

له توضیحات مختصری از هر پارت کد:

- IF: تو اینستراکشن Fetch ، از مازول BR تا اخر میشه برنچ پر دیکتور. بالاش هم میشه اینستراکشن $Fetch$ ، کاری که میکنه اینه که از $Program Counter$ استفاده میکند $Program Counter$ میگه سر $Postage clack$ اگر این $Enable$ بود دان میشه. یه mux داریم که قرار بود بدونیم از چی بخونیم. از $Memory$ و $Branch$ بخونیم یا از Sel بخونیم. اینستراکشن $Branch Predict$ تو این ماکس به مای 2 تا چهار تا داریم. چون یه موقع میخوایم $pc4$ رو بریزی. یه موقع میخوای ادرس $Branch$ بریزی. خط بالاش sel هست. میخوایم اون $Branch Predictor$ داره عمل میکنه، میره با اون ادرسی که داره پیشینی میکنه، یه موقع میگه که فلاش کن. یعنی اشتباه $Predict$ کردم. برو همونجایی که من اشتباه $Predict$ کردم و باید ادامه میدادی، ندادی. پی سی اوت 3 همونه. که درواقع 3 استیج قبل $pc+4$ همون روندی که داشت میرفت رو بره، که تو خط 6 پی سی پلاس 4 مساوی pc_out بعلاوه 4: pc_out خروجی pc . پس خروجی pc میاد بعاوه 4 میشه $pc4$ میسازه. میاد تو $Branch Predictor$ سه مرحله درواقع $Buffer$ میشه. مرحله سومشو میدیم به خروجی؛ اگر فلاش خواستیم بکنیم pc_out_111 رو میذاریم واسه ادرس درستمون تو pc ; خروجیش میشه pc_in ، pc_in رو میذاریم رو خط 10 ($Right\ Enable$)؛ که 15 pc_out تا 0 اش میاد به اینستراکشن مموری؛ خروجیش که اینستراکشن. چه زمانی $Branching$ تو اینستراکشن $Fetch$ موقعی که خط 4 باشد. (یا 1 $Branch$ عه یا 2 $Branch$)

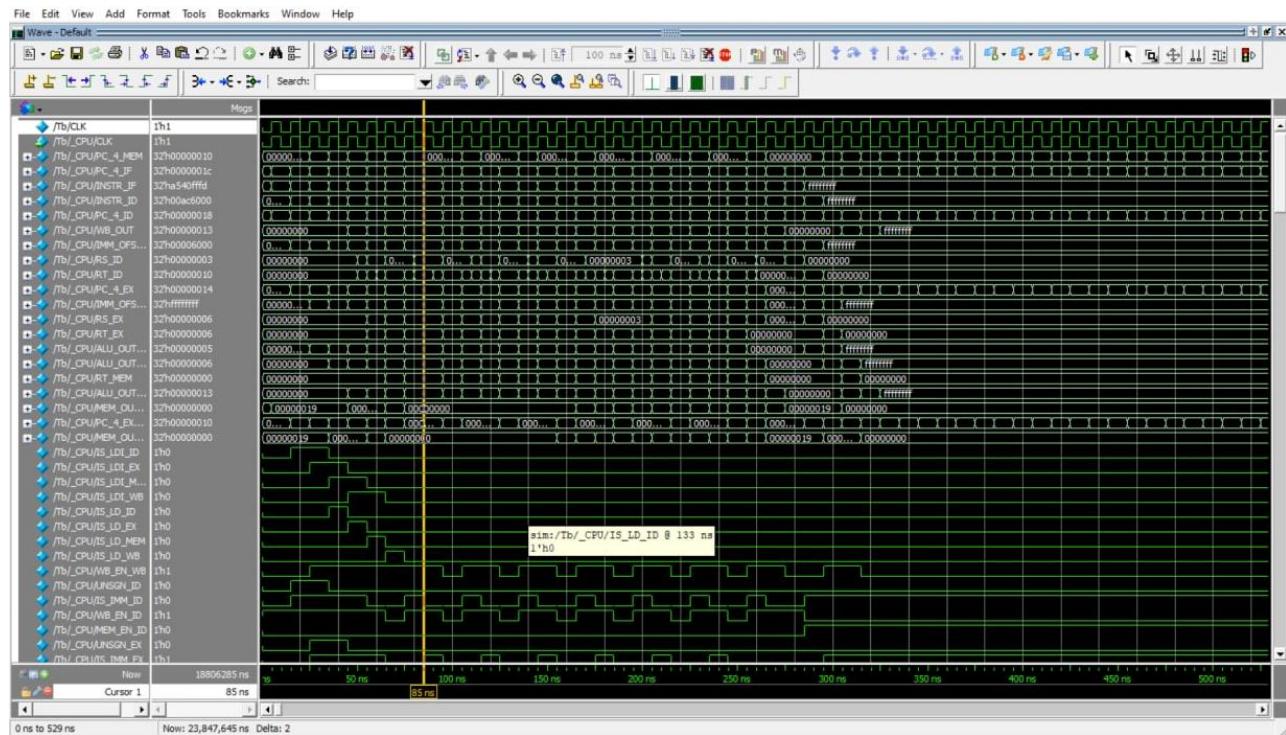
دوتا $Branch$ داشتیم. بعد ادرس $Branch$ رو میسازه، که درواقع این میشه که ضربدر 4 کنیم یا 2 بیت شیفت بدیم به سمت چپ؛ که اونم که $Sign$ بیش که 16 بیت $Sign\ Extend$ کردیم. BR_ADD هم میگه که اگر sel تو 2 عه. قراره که $Branch$ بری و $Predictor$ استفاده کنی. $PC4+BR$ استفاده کن.

- IF_ID: تو $Buffer$ و سطمون، چیزی جز یک رجیستر نیستند که بیشتو 256 بیت گرفتیم. این $Buffer$ های و سطم موقعی که فلاش میشن (ریست میشن)، وقتی ریست میشه همرو صفر میکنه به جز $input$ و out اینکه تو شونم چه مدلیه.

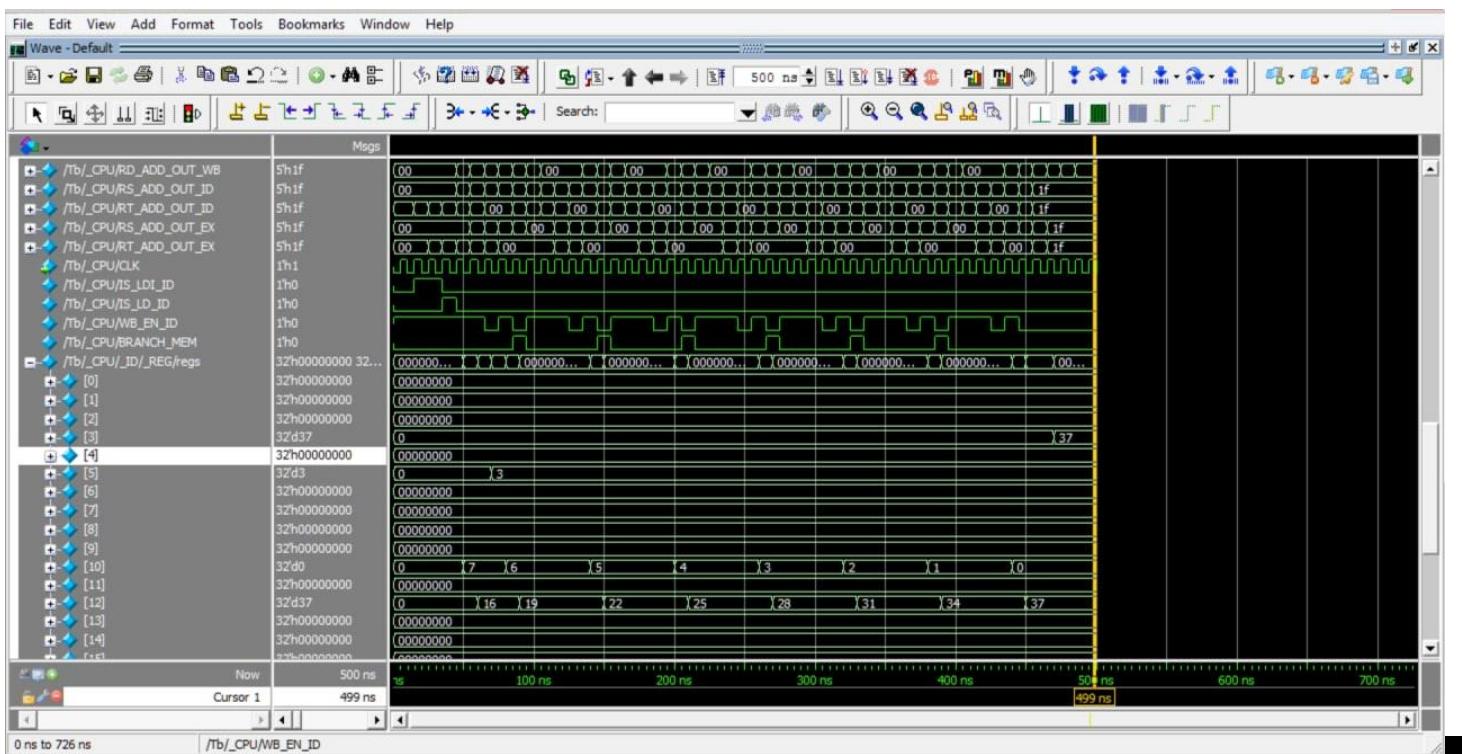
- BUFF: یک پارامتر تعريف کردیم N مساوی 32، که این قابل تغییره همونطور که تو اینستراکشن $Fetch$ دیکود 256 قابل تغییره؛ برای کل چیزها یعنی سر $Postage$ ها اگر دیلی بود، ورودیتو صاف بذار رو خروجی. (دیافت همرو 256 بیت استفاده نمیکنیم از همش) مثلا از 130 تاش استفاده میکنیم یه جا. بعدش مقدار نداره. این سه تا مساوی ینی اینکه اون رو هم درنظر بگیر تو محاسبات. اگر یک واقعی بود، با درنظر گرفتن مقادیر صفر یک و ... اگر یک بود 1 میگذاریم درغیر اینصورت چه صفر بود چه Z بود صفر بذار. (اگر $Enable$ بود)

- ID: توی دیکود، یه سری پارامتر تعريف کردیم. برنج 1 برنج 2 و ... Op_Code رو برداشتیم. آدرس RS آدرس RT از اینستراکشن برداشتیم که همینارو میدیم به خروجی (خط 9 و 10); بعد برای Sign و Unsign ها، اینا اونایین که آنساینده استفاده میکنن (خط 7)
- REG: 32 تا رجیستر داریم که هرکدام 32 بیتن. برا اولیا صفر گذاشتیم. اگر که رایت او مد برو Right رو بریز توی رجیستر با این ادرسی که بهم میده. دوتام ادرس ورودی داره که به ما میده، ما رجیستر هارو بدون دلیل میداریم توی خروجی.
- EX: خط 8 و 9 همون دوتا MUX هست که گفتیم. خط بالاش Branch رو تشخیص میده با توجه به خروجی هایی که از ALU داره. تو خط 7 میگه اگر NE Predictor بودیم، و BR2 Predictor بود، یا EG بودی و 4 بود. BR1 Branch is Branch op_code داریم. (این درواقع Taken هست). خط 4 pc4 Branch رو میداریم آدرس pc4_out Forward کردیم از اینستراکشن Fetch او مده و دوتا شیفت میدیم به چپ. خط آخر هم ALU مون هست.
- ALU: دوتا ورودی داره، که یه ساین داره که با توجه به اون sign، درواقع اگر Unsign بود و بیت بالا مون 1 بود میاد این 1 رو منفی میکنه. و خط 5 به پایین out، add بود sub بود میاد محاسباتشو انجام میده. خط 15 و 16 درواقع Equal Graeter؛ وقتی که داریم از منفی استفاده میکنیم، توی اینستراکشن Execute خط 10 یه Unsign داریم. یعنی علیات رو انجام بده. وقتی انساینه و بیت بالاش منفیه. تو خط 3 اینستراکشن، بیت MS بیش 1 عه منفیه. در واقع تو خط 3 و 4 او مده مثبتش کردیم اون مقدارو. قاعدتا اگر sign هست یه منفی تو ش ضرب میکنیم Unsign میشه.
- MEM: خود Memory هست که 4 تا Memory رو نوشتم بقیه رو صفر گذاشتیم رو تو clk مینویسه؛ RD هم بدون clk هست.
- WB: Right Back Stage :WB که در بالا بهش اشاره کردیم هست.
- IMEM: خط 2 نشون میده که 8 بیتیه Memory، از 0 میریزه تا 16 بیت. 2 به توان 16 تا 8 بیت داریم؛ که با توجه به آپ کد ها، او مده یه کد اسمبلی نوشتم، 7 رو میریزه تو R10، 16 رو میریزه تو R12، از خونه سوم میریزه تو R5؛ بریم تو MEM میبینیم خونه سوم 3 هست (خط 8)؛ درواقع 3 رو میریزه تو R5 و r10 هم از 7 میریزه تا صفر Add immediate صفر نبود، سه تا برو عقب تر، از خط 40 شروع کن.

که بعد میاد وقتی که صفر شده r12 میریزه تو r3، مقدار r0 هم با 14 جمع میکنه میشه 14. خروجی محاسبون 12 که میریزه توی خونه 14؛ که خروجیش در عکس های پایین قابل مشاهده است.



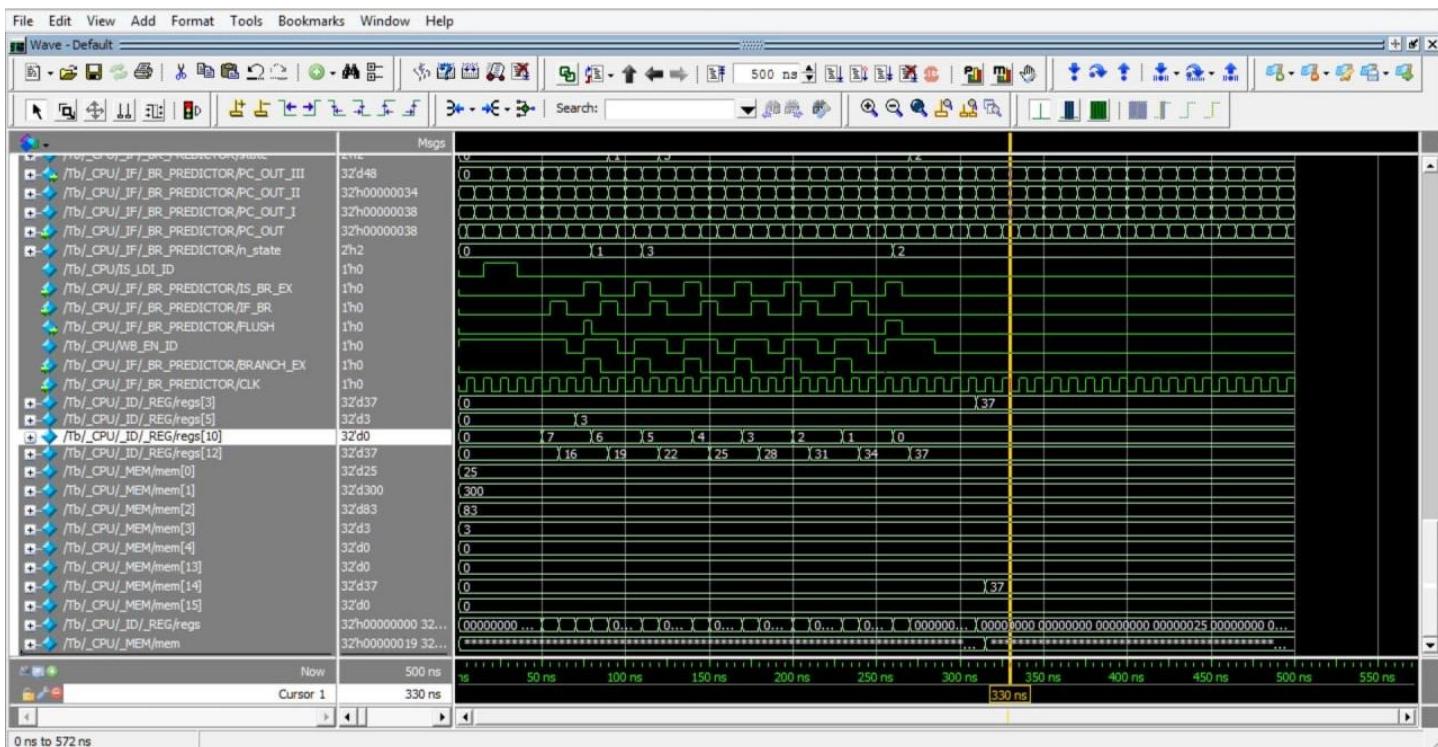
ابتدا در مرحله صفر قرار داریم؛ اگر *Taken* باشد همانجا می‌مانیم. اگر *Not Taken* بود به مرحله یک می‌رویم. در یک در صورت *Not Taken* بودن به صفر بر میگردیم در غیر این صورت به مرحله سه می‌رویم. در مرحله سه در صورت *Taken* بودن همانجا می‌مانیم در غیر این صورت به دو می‌رویم. در دو در صورت *Taken* بودن به سه بر میگردیم در غیر این حالت به صفر می‌رویم. در صورت *Predict* کردن اشتباه به اینستراکشن صحیح خواهیم رفت.



برای اجرای اینستراکشن ها به صورت خارج از ترتیب است.

به این مفهوم که اینستراکشن هایی که نیاز به زمان برای *Execute* دارند برنامه را متوقف نمیکنیم و سراغ اینستراکشن های بعدی رفته و به آنها رسیدگی می کنیم تا نتیجه اینستراکشن مذکور حاضر شود. اما لازم به ذکر است که پایان یافتن اینستراکشن ها به ترتیب می باشد. یعنی تا قبل از اینکه پایان یافتن اینستراکشن فعلی منوط به پایان یافتن قبلی است.

هر چند قبلی در بیست سایکل تمام شود و فعلی در چهار سایکل تمام شود!



افزایش سایز *Buffer* از این جهت موثر است که؛ برای مثال اگر اینستراکشنی بیست سایکل طول بکشد و دستورات بعدی هر کدام یک سایکل طول بکشند می توان بیست دستور را رسیدگی کرد اما اگر سایز *Buffer* هشت باشد آن وقت مجبور به توقف هستیم تا دستور بیست سایکلی تمام شود تا آن خانه آزاد شود. در واقع *Recorder Buffer* یک *Secular Buffer* است با دو *Pointer*: ابتدا و انتها وقتی پر شود دیگر امکان نوشتن نیست تا وقتی یکی خالی و در رجیستر تمام شود.

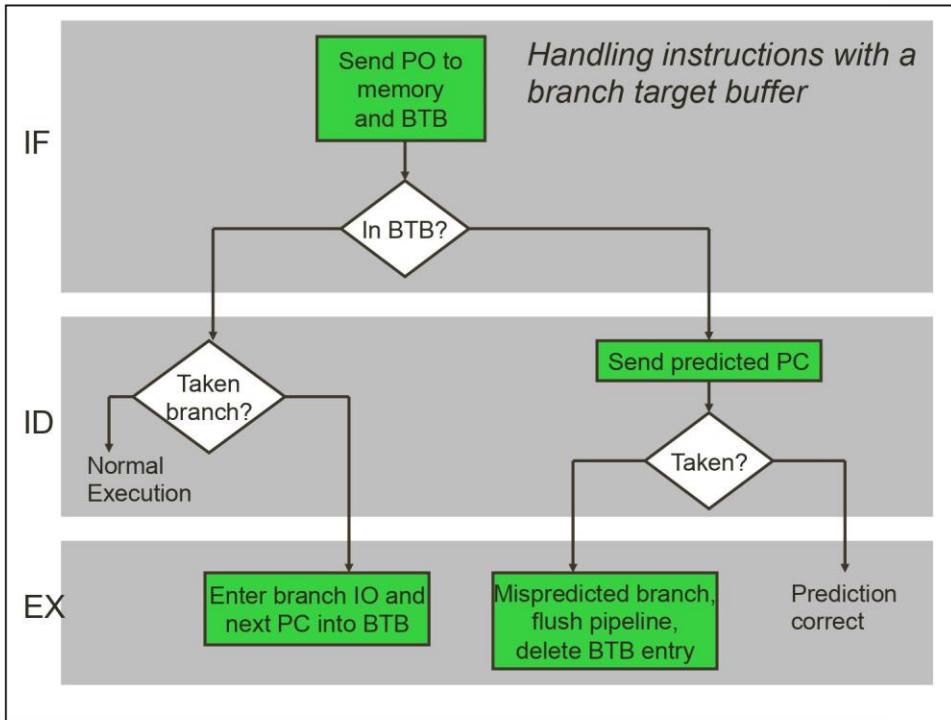
<pre> lui r10,7 lui r12,16 lw r5,r0,3 addi r10,-1 add r12,r5 bne r10,r0,-3 add r3,r12,r0 sw r12,r3,14 </pre>	<pre> mem[0][31:0]=25; mem[1][31:0]=300; mem[2][31:0]=83; mem[3][31:0]=3; for(i=4;i<65536;i=i+1)begin mem[i][31:0]=32'h00000000; end </pre>
--	--

برنامه‌ی تست ما حاصل ضرب سه در هفت را با شانزده جمع می‌کند که برابر سی و هفت است و در رجیستر سه و خانه چهارده حافظه نتیجه را مینویسد. با پیاده سازی تخمین گر پرش برنامه از حدود پانصد نانو ثانیه توانسته به مقدار سیصد و سی نانو ثانیه تقلیل یابد. (یعنی بهبود Performance تا حدود 40%)

ساختار rob: برای پایپ لاین Out of order فرقش با این order این است که، مثلاً پایپ لاین ما اوکی بود. فقط لود دوتا سایکل زمان نیاز داشت. ولی مثلاً فرض می‌کنیم، یک پردازنده‌ای داریم که یک اینستراکشنی دارد که یک مولتی پیکیشن می‌خواهد انجام بدهد. (فرض ۱۲۸ بیت در ۱۲۸ بیت) یک عدد بزرگ را در یک عدد بزرگ می‌خواهد ضرب کند. این طبیعتاً ۱۰ تا، ۲۰ تا سایکل زمان نمیرد، کلی سایکل زمان میرد. حال دو حالت موجود است. یک اینکه می‌ایستیم تا آن مولتی پیکیشن در ۱۰۰۰ سایکل کارش را انجام دهد. حال در این ۱۰۰۰ سیکل ما چندتا کار می‌توانیم انجام دهیم. می‌توانیم اینستراکشن بعدی را انجام دهیم که وابسته به خروجی این نیست. درواقع دستور هارا با تایم بندی خودمان انجام میدهیم. Out of order برای بخشی است که دستورات بعدی را اجرا کرده که پایپ لاینمای سریع تر بشود. اگر همگی یک سیکل بودند مساله این بوجود نمی‌آمد، اما پایپ لانی داشته باشیم که به فرض بالای ۱۰ استیج داشته باشد، این کمک بزرگی می‌تواند باشد.

چند نوع forwarding: یک اینستراکشنی داریم که یک R_s و R_t وجود دارند که درواقع اطلاعات جدید در آن‌ها ذخیره می‌شود. (اطلاعات درحال بروز رسانی می‌باشند). یعنی اینستراکشن قبلی قصد دارد در آن بنویسد. به فرض در اینستراکشن قبلی گفته‌یم، در r_5 مقدار ۱۸ را برویم. بعد r_5 را می‌خوانیم. چون r_5 هنوز حاضر نیست که در اینستراکشن ان را $forward$ کنیم. با یک کپی از دیتا در استیجی که قرار دارد (یا در MEM right back یا در MEM را بخوانیم، باید ان را $forward$ کنیم. با یک کپی از دیتا در استیجی که قرار دارد forward صورت پذیرد یا خیر؛ اینستراکشنی که در مرحله execution می‌باشد، R_s و R_t را استخراج کرده و از دو استیج آخر R مقصود را بر می‌گزینیم. اگر R destination یکی از آنها با یکی از ورودی‌های execution state برابر بود، پس متوجه می‌شویم که اینجا یک hazard داریم.

توضیحات الگوریتم‌های branch prediction: ۶ تا الگوریتم می‌باشد. ساده‌ترین حالت آن زمانی است که ما یک بیت داشته باشیم. (که درواقع حالت قبلی را داراست). به فرض در حافظه not taken بودنمان را حفظ کرده است. می‌بینیم که branch not taken عمل می‌کند. در مرحله execute هم یک فیدبکی از سمت آن دریافت می‌شود. حال ادامه می‌باید و اگر یک branch به حالت taken is بود (و ما متوجه می‌شویم که اشتباه است و not taken باید عمل می‌کردیم) پس اطلاعشان کرده و بعد این را taken is کرده و اینستراکشن‌هایی که طی شده بود را فلاش کرده و به لوکیشنی که باید مراجعه می‌کردیم اما نکرده بودیم می‌رویم. در دو بیت هم که در پژوهه اعمال شده است، در بخش توضیحات کد موجود می‌باشد. از الگوریتم‌های دیگر که global و local هم داریم.



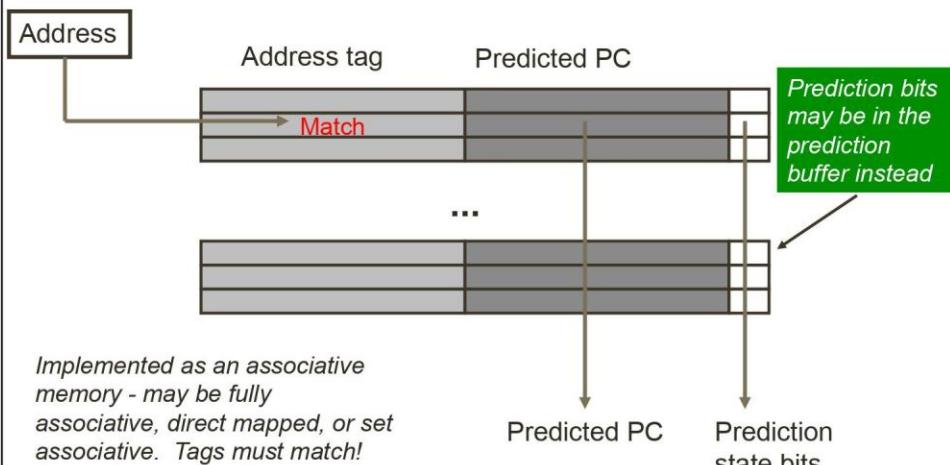
Branch Prediction Summary

- Local - history of a single branch (pattern)
- Global - correlating branches
- Combined - some branches better predicted with global than local and vice versa. Combined can select among both.

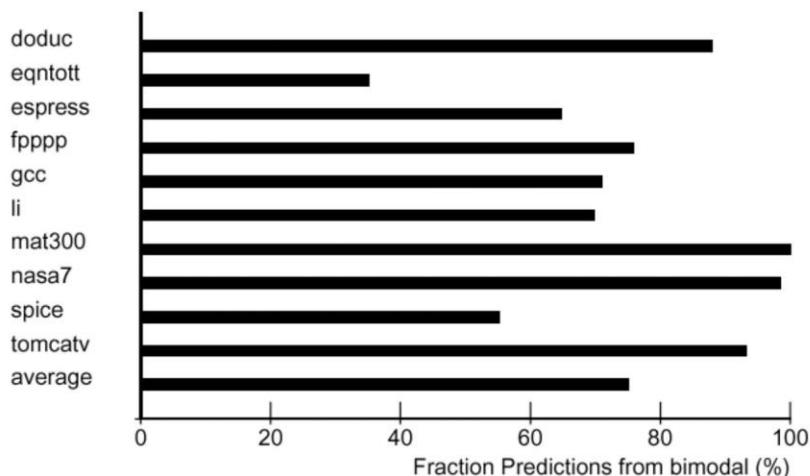
Branch Target Buffer

- Need to know whether the fetched instruction is predicted as a taken branch.
- Unlike BHT, we must tag all entries to ensure the entry corresponds to an actual branch.
- We don't even know if the instruction is a branch since it's not decoded!
- Store only predicted taken branches in BTB
 - May require two tables: One for predicted branch targets and one for the branch predictor.

BTB Implementation



Which Contributes the Most?



Usually bimodal used more, but gshare helps and the predictor is chosen on a per branch basis

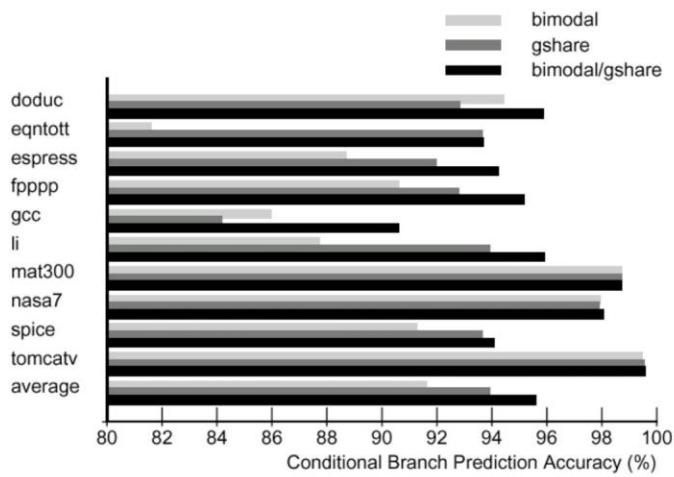
Branch Target Buffer (BTB)

- In DLX, we need the fetch address at end of IF
- Need to know: Undecoded instruction is a branch and what the next PC should be.
- **Buffer to hold next predicted branch target address - “branch target buffer”**
- Essentially, with the branch direction prediction, we can also buffer the predicted target address.

Bimodal/gshare Tournament Predictor

- Branches tend to show either local or global history
- **Bimodal** - use when local history is beneficial
- **Gshare** - use when global history is beneficial
- Adapts to the particular branch by way of the predictor selection mechanism

Tournament Predictor Performance



Tournament predictor always better than either predictor alone

Keeping Track of Predictor Accuracy

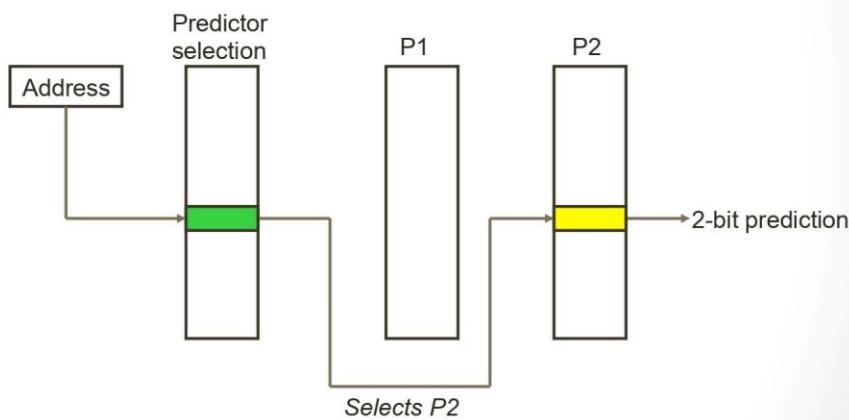
- 2-bit counter incremented/decremented

P1-correct	P2-correct	P1-correct - P2-correct	Action
0	0	0-0 = 0	None
0	1	0-1 = -1	Decrement
1	0	1-0 = 1	Increment
1	1	1-1 = 0	None

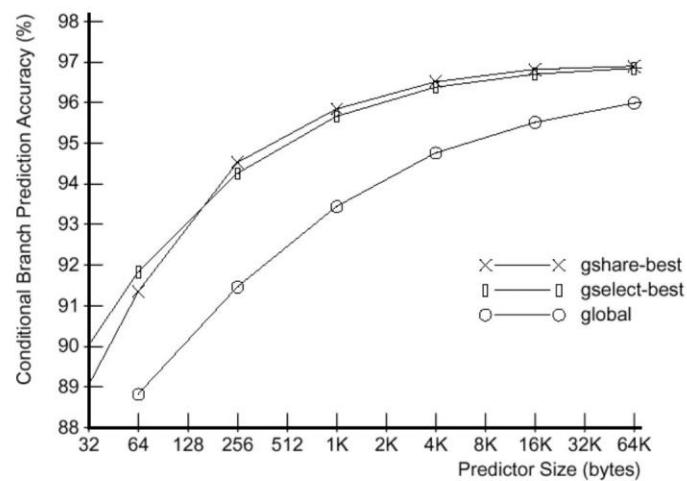
Counter value	Use predictor
00	P2
01	P2
10	P1
11	P1

Selects which predictor table to use for the prediction

Tournament Predictor Implementation



Gshare vs Gselect



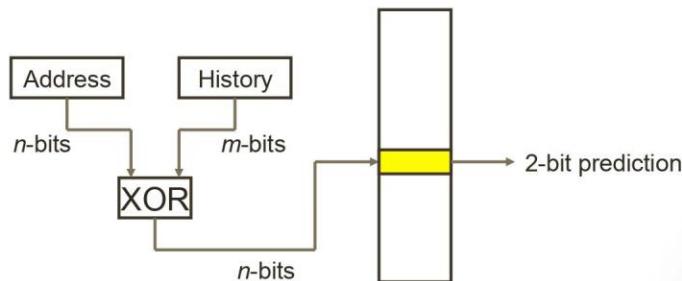
Tournament Branch Predictors¹

- Combine previous schemes into a scheme that has advantages of both
- Select among predictors P1 and P2
- A separate counter array picks among P1 and P2 - i.e., which prediction to use.
- 2-bit saturating counter - counters keep track of which predictor is more accurate

¹also known as “combining predictors”

Global with Index Sharing

- So called “gshare” predictor
- Similar to “gselect” predictor, except the branch address and global history are combined by doing an XOR



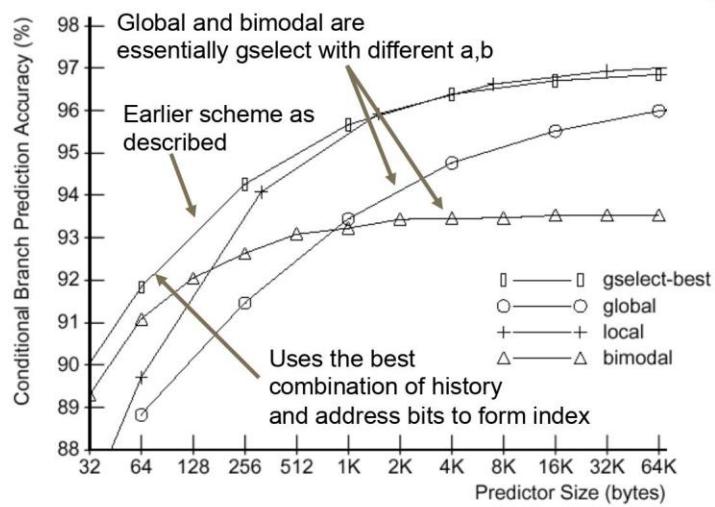
Global History w/Index Sharing

- Hash on the address + global history
- Better able to identify branches

Branch	Global		
Address	History	gselect	gshare
00000000	00000001	00000001	00000001
00000000	00000000	00000000	00000000
11111111	00000000	11110000	11111111
11111111	10000000	11110000	01111111

- Gselect lost the history in the upper four bits

Local vs. Global Select



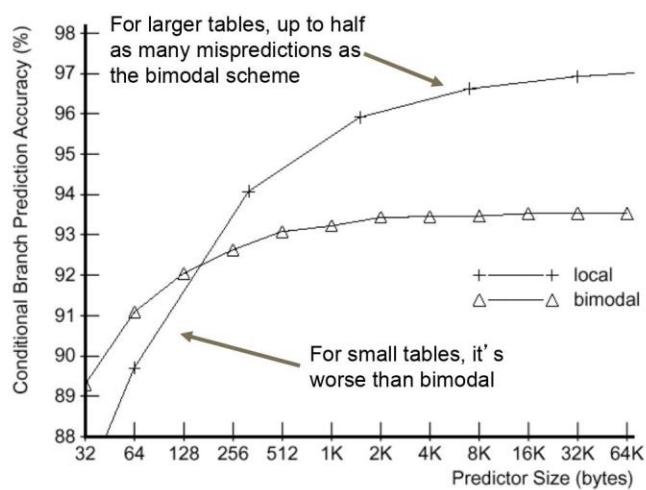
Local vs. Gselect

- Gselect better for < 1KB tables
- Local better for > 1KB tables (but gselect is close)

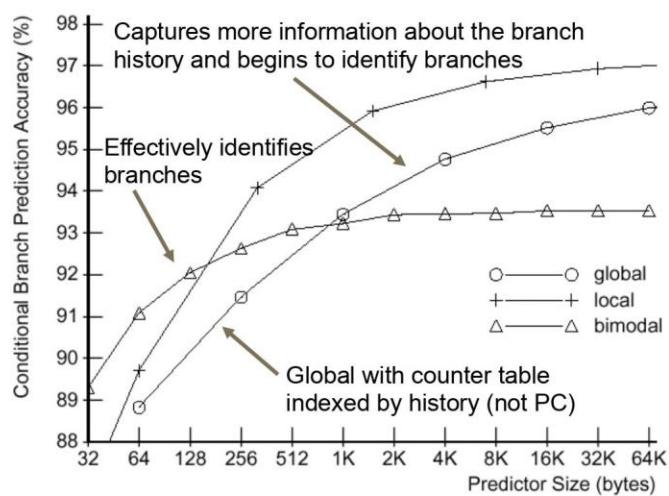
- gselect - storage space for global history is small

- gselect - a single array access
- local - two array accesses
- Thus, gselect potentially faster

Local Branch Prediction Accuracy



Local vs. Global Accuracy



Local Branch Prediction

- Assume some branch executed repeatedly.
- With 3 bits of history and 2^3 counters, the predictor can always predict the branch.
- Each execution has unique history (to index into prediction table)

Shift in 1 on a taken branch to the history

<u>History</u>	<u>History</u>
000 - iteration 0	100
001 - iteration 1	101
010	110 - iteration 4
011 - iteration 2	111 - iteration 3

Contention in Local Predictors

Local predictors can suffer from contention

- (1) History may be a mix of histories from different branches that map to the same history entry
- (2) Conflicts on similar history patterns

- E.g., $(0110)^n$ and $(1110)^n$ map to same entry when branch history entry says “110” .

Local Branch Prediction

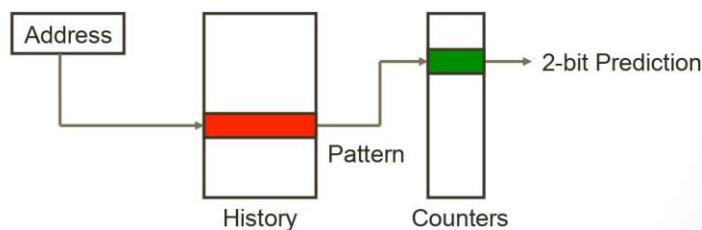
- Consider the loop

```
for (i=1; i<=4; i++) { ... }
```

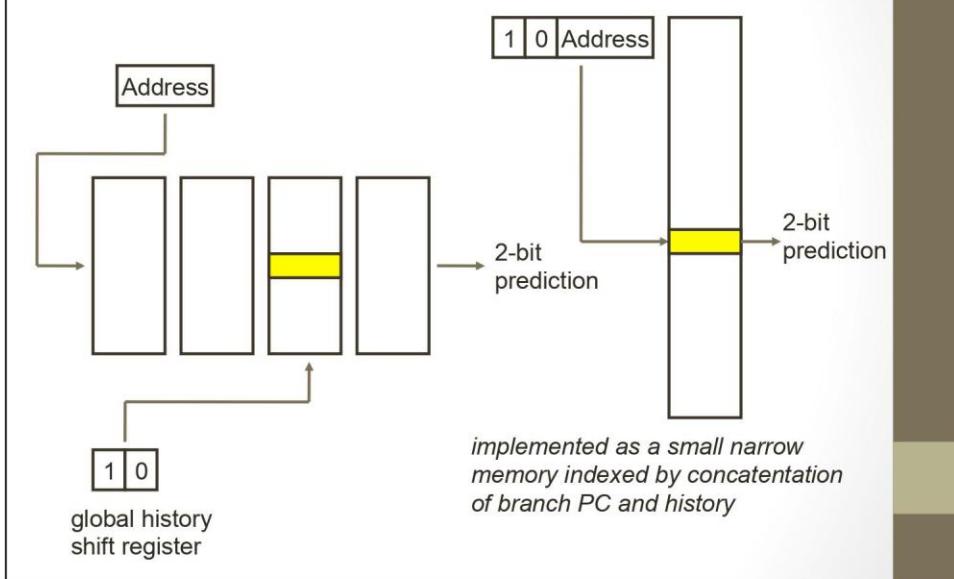
- Loop branch executes with pattern $(1110)^n$
- If we know how the branch has behaved previously, we can predict it.
- Local predictors use the past history of a *particular* branch (unlike the previous scheme - a global predictor)**

Local Branch Prediction

- A two-level history table
- Table 1:** history of recent branches indexed by the low address bits of branch instruction PC
- Table 2:** two-bit branch predictors indexed by the history from table 1



(2,2) Implementation



Trade-off in (m,n) Predictor

- m bits used to select predictor entry
- $m = a + b$ bits
 - a is number of address bits
 - b is number of history bits
- We want enough address bits that each branch is reasonably well identified, along with an increasing number of history bits.
- Bimodal is $b=0, a=m$
- Global history is $b=m, a=0$

1-Bit Pred., 1-Branch Correlation

d	B1	B1	New B1	B2	B2	New B2
	predict	actual	predict	predict	action	predict
2	NT/NT	T	T/NT	NT/NT	T	NT/T
0	T/NT	NT	T/NT	NT/T	NT	NT/T
2	T/NT	T	T/NT	NT/T	T	NT/T
0	T/NT	NT	T/NT	NT/T	NT	NT/T

d alternates between 2 and 0

Predictors for B1 and B2 are initialized to not taken (NT/NT)

What happens with the branch predictions???

Notation: prediction if last branch not taken/prediction if last branch taken

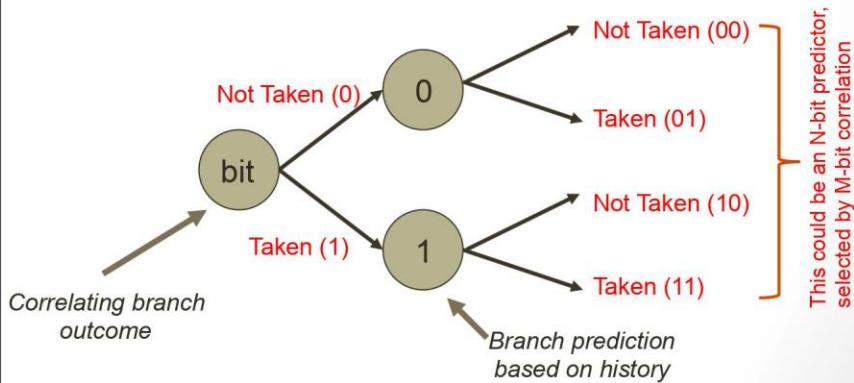
Only the first iteration is mispredicted!

Prediction with Correlation

- (m, n) predictor
 - m bits of correlation
 - n -bit predictor for branch
 - last m branches (2^m) each with an n -bit predictor
- Implementation: Global history with selected address bits (so called “gselect”)
 - m -bit shift register holds outcome of last m branches
 - BHT indexed by $m:low(PC)$
 - BHT can also be indexed just by m (global history prediction)

Prediction with Correlation

- With 1-bit of correlation, each branch predictor has a prediction for:
 - previous branch taken
 - previous branch not taken



1-Bit Pred., 1-Branch Correlation

d	B1 predict	B1 actual	New B1 predict	B2 predict	B2 action	New B2 predict
2						
0						
2						
0						

d alternates between 2 and 0

Predictors for B1 and B2 are initialized to not taken (NT/NT)



What happens with the branch predictions???

One-Bit Predictor

d	B1	B1	New B1	B2	B2	New B2
	predict	actual	predict	predict	action	predict
2	NT			NT		
0						
2						
0						

d alternates between 2 and 0

Predictors for B1 and B2 are initialized to not taken (NT)

What happens with the branch predictions???

One-Bit Predictor

d	B1	B1	New B1	B2	B2	New B2
	predict	actual	predict	predict	action	predict
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT

d alternates between 2 and 0

Predictors for B1 and B2 are initialized to not taken (NT)

What happens with the branch predictions???

All branches are mispredicted!

Correlating Branches

- Branch history can lead to better decisions

```

if (aa==2)           SUBUI R3,R1,2
    aa=0;          BNEZ  R3,L1   ← B1
if (bb==2)           ADD    R1,R0,R0
    bb=0;          L1:   SUBUI R3,R2,2
if (aa!=bb) { ... } BNEZ  R3,L2   ← B2
                    ADD    R2,R0,R0
                    L2:   SUBU  R3,R1,R2
                    BEQZ  R3,L3   ← B3
    
```

If B1 and B2 both taken, then B3 is probably not taken (110)

If B1 and B2 both not taken, then B3 is taken (001)

Correlating Branches

```

if (d == 0)           BNEZ  R1,L1   ← B1
    d=1;          ADDI   R1,R0,1
if (d == 1) { ... }  L1:   SUBUI R3,R1,1
                    BNEZ  R3,L2   ← B2
                    ...
                    L2:   ...
    
```

<u>d</u>	<u>d==0?</u>	<u>B1</u>	<u>d before B2</u>	<u>d==1?</u>	<u>B2</u>
0	Yes	Not taken	1	Yes	Not taken
1	No	Taken	1	Yes	Not taken
2	No	Taken	2	No	Taken

If B1 is not taken, then B2 is not taken (00).

BHT Performance

- “Bimodal prediction” works well - branches fall into one of two camps: **taken or not taken**
- Accuracy isn’t enough - frequency also important
 - More frequent branches, the better accuracy required
- Integer codes (e.g., gcc, eqntott, espresso) may have very frequent branches
- With more ILP, accuracy (with frequency) becomes vitally important.

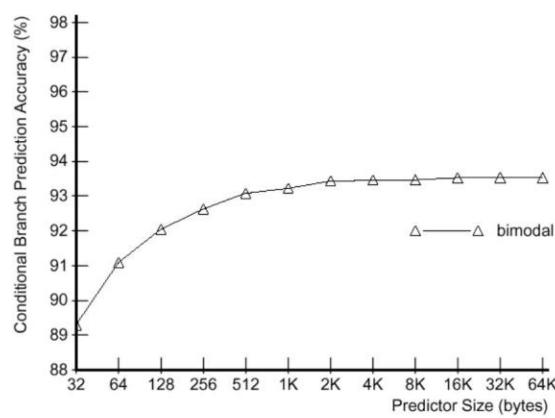
Improving on BHT

- Even with infinite table size - accuracy is not much improved over 4096 entries
 - Conflicts in the table isn’t the problem
- Increasing bits per entry also does not help.
- **Problem:** *BHT uses only recent local history of a branch to predict future (not pattern based)*
- **Solution:** *Look at global history of other branches in making a prediction about the current one.*

BHT Implementation

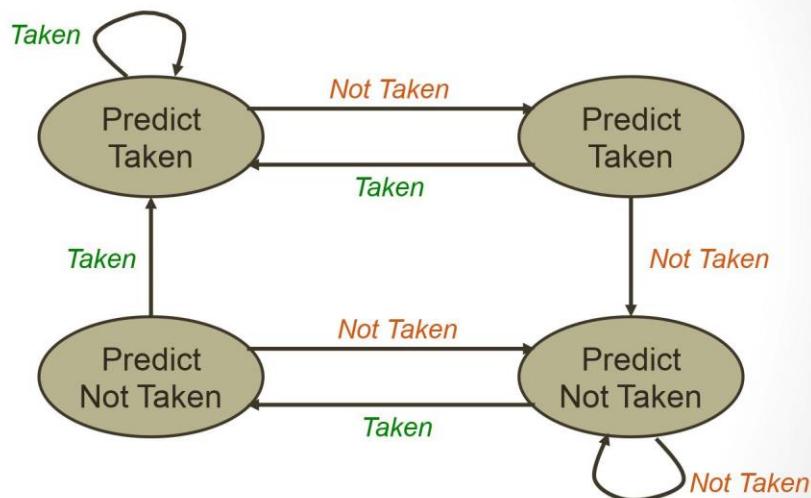
- A small cache accessed during IF
- Counter (two bits) attached to each cache line
- If branch predicted taken, fetch begins from target *as soon as target PC known*
- In DLX, the branch outcome and target are known at same time - no advantage for such a simple pipeline

Two-Bit Prediction Accuracy



Prediction accuracy for SPEC' 89. Accuracy approaches that of an infinite table size.

Two-Bit Prediction



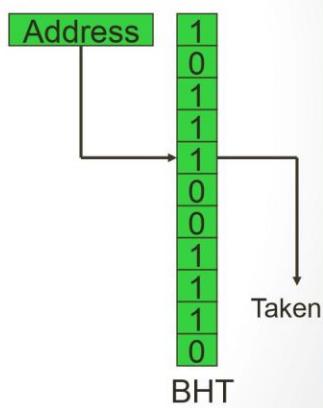
Two-Bit Saturating Counters

- Two-bit scheme may be implemented as a saturating counter
 - MSB indicates branch prediction
 - Increment on a taken branch
 - Decrement on a not-taken branch
- Specialized case of n -bit saturating counter
 - Values 0 to $2^n - 1$,
 - Don't increment/decrement past maximum/minimum value
 - Predict taken when counter > one half maximum value
 - Two-bit scheme works nearly as well as larger number of bits

State	Description
00	No taken branches, initial
01	One taken branch
10	Two taken branches
11	Three taken branches

Branch History Table (BHT)

- Memory indexed by lower portion of address of branch instructions (**a local scheme**)
- A single bit indicates direction
 - Previously: 1=taken, 0=not taken
 - Previous direction is current prediction
- On a branch, record the correct outcome of the branch
- Multiple branches may map to the same table entry



Two-Bit Prediction

- Previous scheme - one-bit prediction
 - Consider a loop: even with all branches taken, there will be two mispredictions (one at the beginning and one when exiting the loop)
- Extend to two-bit scheme
 - A prediction must be inaccurate twice before it's changed

Branch Prediction

- Tackles problem of stalls from control dependencies
- Vital for multiple issue architectures
 - Branches arrive up to N times faster when issuing up to N instructions per clock cycle
 - Relative impact increases with lower potential CPI (from Amdahl's Law)
- Hardware based branch prediction
 - Dynamically predict **outcome** and **target** of branches
 - Uses run-time knowledge of branch behavior history

Branch Prediction

- Effectiveness dependent on
 - Prediction accuracy (how many predictions were correct)
 - Latency of correct predictions
 - Penalty of incorrect predictions
- Prediction accuracy and latencies depend on
 - Structure of pipeline
 - Type of predictor
 - Misprediction recovery strategies
- Local and global schemes
 - Local: predicts based on the current branch
 - Global: predicts based on previous related branches