

ארכיטקטורת מחשבים

שיעור 1: 1. מגמות בשימוש במחשבים ובטכנולוגיות:

- כשמגדירים Instruction Set צריך לקחת בחשבון את העלייה בזיכרון שבו משתמשות התוכניות. במערכות מחשב נחוצות 3 טכנולוגיות בסיסיות:
1. *Integrated Circuit Logic Technology*: מהירות ה-devices וה-transistor count גדלים במהירות.
 2. *Semiconductor DRAM*: cycle time משתפר (קטן) באופן איטי מאוד.
 3. *טכנולוגיות דיסק מגנטי*: disk density גדלה במהירות. זמן גישה משתפר באיטיות.

2. ביצועי המחשב:

BogoMips: מידה של ביצועים במע' Linux. ה-BogoMips מדרג את מהירות המע' - כמה פעולות עושים בכל שנייה.

$$(1) \frac{MHz}{BogoMips} = CPI$$

המטרה: הקטנת ה-response time של המחשב.

$$(2) \frac{ExTime(Y)}{ExTime(X)} = \frac{Performance(X)}{Performance(Y)}$$

חוק Amdahl:

חוק Amdahl משמש להערכת השיפור בביצועים כתוצאה משימוש במצב ביצוע אחר לביצוע תת קבוצה של פונקציונליות מע' המחשב.

Speedup: היחס של הביצועים (performance) לאחר השיפור לעומת הביצועים לפני השיפור.

$$(3) F_e = \frac{T_e}{T_{old}}$$

- T_e : הזמן של החלק שאני רוצה לשפר (לפני השיפור).
- F_e : החלק היחסי בזמן הביצוע של החלק אותו נרצה לשפר במכונה המקורית.
- S_e : השיפור עקב ביצוע ה-enhanced execution.
- T_{old} : זמן הביצוע המקורי.
- T_{new} : זמן הביצוע החדש.

$$(4) T_{new} = T_{old} \left((1 - F_e) + \frac{F_e}{S_e} \right)$$

$$(5) Speedup = \frac{T_{old}}{T_{new}} = \frac{1}{(1 - F_e) + \frac{F_e}{S_e}}$$

סה"כ זמן CPU של תוכנית:

$$(6) T_{CPU} = \frac{CPU\ Cycles}{Clock\ Rate} = (CPU\ Cycles) \cdot (Clock\ Cycle\ Time)$$

$$(7) T_{CPU} = \frac{IC \times CPI}{Clock\ Rate} = IC \times CPI \times Clock\ Cycle\ Time$$

כלומר, עמ"נ לשפר את הביצועים צריך להגדיל את אחד הגורמים במשוואה (6).

$$(8) CPU\ Cycles = \sum_{i=1}^n CPI_i \times IC_i$$

דוגמה:

אם $S_e = 1$, כלומר לא משפרים שום דבר ← מקבלים $Speedup = 1$.
 אם $S_e = \infty$, כלומר מצליחים לעשות משהו בחינם, זה לא אומר שה- $Speedup$ יהיה אינסופי.

שימוש נוסף לנוסחת Amdahl: מעבדים מקבילים:

אם יש כמה בעיות שניתן לעשות במקביל וכמה שניתן לבצע סדרתית, ניתן לעשות Speedup רק לחלק שניתן לעשות במקביל. במקרה זה:
 F_e : החלק היחסי של הבעיה שהוא מקבילי.
 S_e : מספר המעבדים שמוסיפים.

מה כדאי לשפר כדי להקטין את זמן הביצוע?

Clock Rate	CPI	Instruction Count	
		X	תוכנית
	X	X	מהדר
	X	X	Instruction Set
X	X		ארגון
X			טכנולוגיה

שיעור 2: 1. ארכיטקטורת DLX

CISC: מחשב עם פעולות מסובכות: מאופיין בפקודות מחשב קצרות, פעולות של מחרוזות, סט פקודות עשיר (VAX). **בעיה:** ביצועים, overhead, מעט אוגרים.

RISC:

ל-DLX יש 3 סוגי פקודות. כל פקודה היא בגודל אחיד – 32 ביטים ← פענוח מהיר.
 מספר האוגרים הוא די גדול – 32. כל אוגר מיוצג ע"י 5 ביטים.
 התחלת פקודה היא תמיד **כפולה של 4**. כל ה-data נכתב/נקרא מהזיכרון בכפולה טבעית (כפולה של 4).

I-Type Instruction:

31	26	25	21	20	16	15	0
OP-Code		rs			rd		immediate

R-Type Instruction:

31	26	25	21	20	16	15	11	10	6	5	0
OP-Code		rs			rt		rd		shamt		func

J-Type Instruction:

31	26	25	0
OP-Code		jump offset	

פקודות לדוגמה:

Data Transfer: lb, lh, lw, ld, sd, movd.

ALU Instructions: add, addi, addui, sub, mult, div, and, or, xor, lhi, sll, srl, sra, seq, sne, slt, sgt.

Control Instructions: beqz, bnez, j, jr, jal, jalr, trap, rfe.

FP Instructions: addd, addf, subd, divd, multd, multf, cvtd2f, cvtf2d, eqd, ltd, led.

lw r5, o(r3) // $r5 \leftarrow \text{addr}[r3 + \text{offset}]$.

sw 8(r3), r5 // $r5 \leftarrow [r3 + 8]$.

דוגמה:

```
int a[5], *pa;
pa = &a[2]; // 4 של כפולה של 4
*pa = 5;
pa = 2 + (char *)&a[2]; // 4-ב ולא ב-2
*pa = 5; // segmentation - עפים ולכן עפים
```

העובדה שהפקודות הן aligned חוסכת בסיבוכיות.
 Intel מאפשר גם unaligned ← יותר מסובך ואיטי.

nop: בד"כ מופיע אחרי branch/jump. RISC בד"כ עובדים על pipeline ובגלל שהם עובדים מהר צריך להכניס זמן "לנשום". ב-branch צריך להגיע לקטע אחר בתוכנית, אולם ה-pipeline מלא בדברים אחרים. לכן משתמשים ב-nop ישר אחרי ה-branch כדי לא להגיע למקומות שלא רצינו.

עמ' 4 – שיעור 2 – מבנה ה-Datapath.

עמ' 4-5 – שיעור 2 – תרגום תוכנית C לאסמבלר

שיעור 3: ארכיטקטורת DLX

עמ' 2 – שיעור 3 – Datapath של ה-DLX מאפשר לכל פקודה להתבצע לאחר 4 או 5 cycles. בגלל בעיה של סנכרון מחלקים את זמן המחשב לפעימות שעון.

יש 5 שלבים:

Stage	Memory Reference	Register – Register ALU	Register – Imm ALU	Branch
IF		IR ← MEM[PC] NPC ← PC + 4		
ID		A ← Regs[IR _{21...25}] B ← Regs[IR _{16...20}] Imm ← (IR ₁₆) ## ¹⁶ IR _{0...15} //sign extension		
EX	ALUOut ← A + Imm	ALUOut ← A func B	ALUOUT ← A op Imm	ALUOUT ← NPC + Imm Cond ← A op 0
MEM	(for loads) LMD ← Mem[ALUOut] (for stores) Mem[ALUOut] ← B			if (cond) PC ← ALUOut else PC ← NPC עבור branch עם cond עוברים לפלט של ה-ALU (הפקודה הבאה המחושבת). אחרת ל-4 PC = NPC +
WB	(for loads) Regs[IR _{16...20}] ← LMD	Regs[IR _{11...15}] ← ALUOut	Regs[IR _{16...20}] ← ALUOut	

IF – Instruction Fetch: קריאת הפקודה מהזיכרון.
ID – Instruction Decode / Register Fetch: שליפת הערכים מהאוגרים.
EX – Execute / Address Calculation: ביצוע פעולות אריתמטיות.
MEM – Memory Access: כתיבה/קריאה לזיכרון.
WB – Write Back: כותבים אוגר.

שיעור 4: Pipelining

pipelining doesn't help latency of a single task, it helps throughput of an entire workload.

עמ' 3 – שיעור 3 – Pipeline

Pipeline: יצירת חפיפה בין דברים במהלך הביצוע. לוקחים את מה שלוקח הכי הרבה זמן וזו תהיה היחידה הבסיסית.

חסרון: יוצר latency ← זה לא טוב עבור RT.

יתרון: ה-throughput יגדל.

בכל מלבן (latch) שומרים הנתונים המקומיים הנוכחיים והם זורמים מאחד לשני. הפלט של ה-latch קבוע. ברגע שיש אירוע (סוף פעימת שעון) אזי הקלט באותו רגע נהפך לפלט ועד סוף פעימת השעון הקלט לא משתנה.

superscalar: בתוך ה-chip יהיה יותר מ-ALU אחד.

Hazards

עמ' 5-4 – שיעור 3 – Hazards

ב-pipeline אידיאלי בלי hazards הכל עובר חלק ולכן בממוצע תתבצע פקודה אחת כל פעימת שעון אחת.

3 סוגי מכשולים:

1. מבני (Structural)

כאשר יש זיכרון משותף לנתונים ולפקודות ← **Structural Hazard**, שכן בכל פעימת שעון צריך לקרוא פקודה מהזיכרון ← בכל פעימת שעון ה-memory port תפוס ובפעימת השעון ה-4 יש גישה לזיכרון (קריאת data) ואילו יש גם גישה לכתיבה של ה-job ה-4 ← אין מספיק משאבים באותה פעימת שעון כדי לבצע את הפעולה.

פתרון: stalls: נשהה את ביצוע הפקודה ה-4 כל פעם בפעימה אחת.

הפתרון עובד שכן רק כ-20% מהגישות לזיכרון הן לכתיבה, ואילו בשאר הפעמים הגישה היא רק לקריאה. כלומר יש התנגשות רק כשיש load מול store וזה מתבצע רק בכ-20% מהפקודות.

הערה: במחשב שלנו המכשול הזה לא קורה בד"כ בגלל שיש cache נפרד לנתונים ולפקודות.
2. נתונים (data):

כאשר צריך להשתמש בערך של אוגר לפני שהערך הזה חושב ← **Data Hazard**.
 לדוגמה:

```
add r1, r2, r3 // the value of r1 is computed at the 3rd cycle, but only written in the 5th cycle (WB).
sub r4, r1, r3 // need to use r1 in the 3rd cycle.
```

עמ' 3-1 – שיעור 4 – Data Hazards

נשים לב שהערך של r1 ידוע כבר בסוף הפעימה ה-3, אולם הוא עוד לא נכנס לתוך האוגר – הערך שלו עוד ב-ALU. ניתן לקצר את ה-datapath.

הפתרון: forwarding: הפלט של ה-ALU נכנס ל-mux שלפניו ← בסוף הפעימה ניתן לקרוא את הערך המעודכן (פתרון ללא stalls).

הפתרון טוב עבור חישובים אריתמטיים באוגרים, אולם לא עבור פקודות load. למשל:

```
lw r1, 0(r2) // the value of r1 is known at the end of the 4th cycle.
sub r4, r1, r3 // the value of r1 is needed at the beginning of the 3rd cycle.
```

כאן r1 נוצר רק בפעימה ה-4 ולא ע"י פקודת ALU בפעימה ה-3. אי אפשר לפתור את זה ע"י forwarding שכן הערך עוד לא קיים לפני החישוב שבו הוא נחוץ.

פתרון: stall של פעימת שעון אחת עבור כל הפקודות.

עמ' 4-3 – שיעור 4 – Control Hazards

Control Hazards: יש פקודת branch. הבדיקה האם התנאי לקפיצה מתקיים או לא נעשית רק בפעימה ה-4. מה עושים עד אז? מחשבים כרגיל את שאר הפקודות? שמים 3-stage stalls?

פתרון 0: נעשה השהייה עד שנדע לאן צריך לקפוץ.

פתרון 1: "מנחשים" שה-branch לא מתקיים (predict branch not taken) ולכן ממשיכים את הביצוע הרגיל. אם בזמן הבדיקה של ה-branch רואים שהוא צריך להתקיים אזי מפסיקים את הביצוע של הפקודות, ומבצעים את הקפיצה (אין כאן delay slot). כיוון שבפעימה ה-3 נדע אם צריך לקפוץ או לא, אזי אם צריך לקפוץ נבטל את הפקודות שביצענו – בעצם צריך לבטל רק את מה שב-pipeline שכן אף נתון עוד לא נכתב, עוד לא שינינו שום דבר (שכן עוד לא היה Write Back או כתיבה לזיכרון). השינוי יעשה ע"י write-back לאוגר 0 (שקול ל-nop). אבל לעומת זאת, הפסדנו 3 פעימות שעון.

כיוון שבממוצע כל פעולה חמישית/שישית היא branch אזי הפסד של 3 פעימות שעון כל פעם מכניס בעצם stall בפועל לתוכנית. נרצה להקטין את מספר ה-stalls שנובעים מ-branch.

2 פתרונות:

- שינוי במחשב:** מוסיפים ALU נוסף שתפקידו: לחשב את יעד הקפיצה (PC + 4) + offset. כעת, במקום לחכות 3 פעימות שעון נחכה רק פעימה אחת. אם ה-branch מצליח אזי יש רק השהייה של פעימת שעון אחת. **חסרון:** פתרון מסובך ודורש יותר חומרה.
- branch delay slot:** ברוב מחשבי RISC תמיד מבצעים פעולה, שבין כה וכה צריך לבצע, ישר אחרי ה-branch.

שיטה נוספת: ננחש אם צריך לקפוץ לפי ה-branch target address cache-
 מבנה ה-cache:

address הכתובת של הפקודה	target החישוב
-----------------------------	------------------

ברגע שרואים branch בודקים אם ה-pc מופיע ב-cache ואם כן יודעים מה כתובת הקפיצה (לפי ה-cache) ולא צריך לחשב עוד פעם. לרוב, חוזרים על לולאות ולכן זוכרים את ה-branch ולא צריך לחשב כל פעם מחדש.

branch prediction: יש cache שבו זוכרים מה עשית בפקודה הזו בפעם האחרונה. ההנחה היא שבאופן סטטיסטי יש התאמה בין אם קפצת בפקודה הזו קודם ובין אם תקפוץ בה עכשיו ולפי זה נבחר predict taken או predict untaken.
בפנטיום זוכרים היסטוריה של יותר מפקודה אחת.

שיעור 5: Exceptions

קשה יותר לטפל ב-exceptions במהלך התוכנית בארכיטקטורה עם pipeline.
precise exceptions: אם ה-exceptions גורמות לכל הפקודות שלפני הפקודה שגרמה ל-exceptions להסתיים אולם כל הפקודות שלאחר הפקודה ה"ל לא תושלמנה. [ניתן לשחזר במדויק מה שהיה]

imprecise exceptions: אם ההגדרה של precise exceptions לא מתקיימת (למשל: במחשבי DEC's Alpha).
[למשל: overflow – לא יודעים איך קרה, קשה לעשות debugging]

סוגי ה-exceptions שעשויים להיות ב-DLX pipeline:

Pipeline Stage	Possible Exception
IF	Page fault on fetch = כשצריך להביא קטע מהתוכנית לזיכרון כשצריך אותו, אבל הוא עדיין לא שם. Segmentation fault = מקום לא חוקי - עברו - מקום לא חוקי = Memory protection = execute only text שהוא לקרוא text למשל, ניסיון לקרוא text שהוא execute only
ID	Undefined or illegal op-code. בקריאה של האוגרים עצמם אין exceptions. יש exceptions אם ה-op-code הוא לא חוקי.
EX	Arithmetic exception.
MEM	Page fault on data Segmentation fault Memory protection
WB	None.

עמ' 4-6 - שיעור 5:

Loop Unrolling:

ניתן להשתמש ב-unrolling עמ"נ להגדיל את המספר הממוצע של הפקודות המבוצעות בפעימת שעון אחת. הרעיון ב-loop unrolling: משכפלים את גוף הלולאה מספר פעמים במטרה להקטין את האחוז של מספר הפעמים שמקדישים ל-"loop housekeeping" וכך מספקים יותר מרחב לתזוזה של יחידות הקוד עמ"נ להיפטר מ-stalls בהם לא עושים כלום.
ע"פ loop unrolling ניתן למצוא יותר אפשרויות למקביליות (טוב ל-pipeline).

שיעור 6: Instruction Level Parallelism:

אם ברצוננו לבצע קוד מכונה באופן סדרתי ומהיר צריך לנסות למצוא פקודות שיכולות להתבצע בלי pipeline hazards. ניתן לנסות גם לבצע פקודות מסוימות סימולטנית (במקביל). אולם צריך לשים לב ל-*dependencies* בין הפקודות שמונעות מכמה פקודות להתבצע במקביל.

קיימים 3 סוגי תלויות שמפריעות ל-ILP:

- Data Dependencies:** פקודה j היא data dependent בפקודה i אם אחד מהתנאים הבאים מתקיים:
א. פקודה i מפיקה תוצאה שפקודה j משתמשת בה.
ב. או: פקודה j היא data dependent בפקודה k, ופקודה k היא data dependent בפקודה i.
- Name Dependencies:** כאשר 2 פקודות משתמשות באותו אוגר או באותו מקום בזיכרון (name), אבל אין זרימת נתונים (flow of data) בין הפקודות שה-name הזה משוייך להן.
ע"פ register rewriting משנים שמות אוגרים ש"מתו" כך שניתן לבצע דברים במקביל.
- Control Dependencies:** תלות בקרה קובעת את סדר הפקודות בהתאם לפקודת branch. באופן כללי, יש לשמור על תלויות של בקרה.

Basic Dynamic Branch Prediction:

ככל שרוצים לנצל יותר ILP, כך תלויות הבקרה מהוות גורם מעכב יותר ויותר. לרוב פקודה אחת מתוך 6 פקודות היא פקודת branch.
שיטת ה-dynamic branch prediction הפשוטה ביותר היא *branch prediction buffer* או *branch history table*: קטע זיכרון קטן שהאינדקסים בו הם הביטים התחתונים של כתובת ה-branch של הפקודה. הזיכרון מכיל ביט שמציין האם ה-branch הזה בוצע לאחרונה או לא. אם נאחסן יותר ביטים ניתן לקבל חיזוי יותר טוב. ברוב המעבדים כיום מאחסנים כ-3-4 ביטים.
ה-branch target buffer משמש לשמור את הכתובת של ה-branch כדי שלא תהיה השהייה בשביל החישוב כל פעם.

עמ' 2-3 - שיעור 6: I32 Performance:

Superscalar:

במקום לקרוא 32 ביט קוראים 64 ביט.
בכל פעימת שעון מכניסים 2 פקודות ל-pipeline ומנסים לבצע אותן – קל למתכנת, קשה למכונה.
ב-superscalar ניתן לקחת קוד כתוב ולבצע את זה ללא שינוי בתוכנה (ע"פ מציאת פקודות בלתי תלויות וביצוען) ← implicit parallelism. *יתרון*: לא צריך מהדר חדש.

VLIW (Very Large Instruction Word): המהדר עושה את זה.

לוקח פקודה ארוכה (למשל: 128 ביטים), מחלק את הפקודה לשדות ומבצע את הפקודה ← צריך מהדר חדש או מהדר שיודע לעשות אופטימיזציה ויודע מה ניתן לבצע במקביל – explicit parallelism.

בשניהם ה-CPI קטן מ-1 ← מבצעים יותר מפקודה אחת בבת אחת = מקביליות.

:Pentium Pro - ראשון

The goal: exceed the performance of the 100MHz Pentium processor.

The Pentium processor’s pipelined implementation uses **5 stages** to extract high throughput. The Pentium Pro processor moves to a decoupled, **12-stage**, superpipelined implementation, trading less work per pipestage for more stages (Pentium 4 is **20-stage** pipelined). The Pentium Pro processor reduced its pipestage time by 33%, compared with a Pentium processor, which means the Pentium Pro processor can have a 33% higher clock speed than a Pentium processor.

The Pentium processor’s superscalar microarchitecture has an ability to execute 2 instructions per clock. The new approach used by Pentium Pro processor removes the constraint of linear instruction sequencing between the traditional “fetch” and “execute” phases, and opens up a wide instruction window using an instruction pool. This approach allows the “execute” phase of the Pentium Pro processor to have much more visibility into the program’s instruction stream so that better scheduling may take place. It requires the instruction “fetch/decode” phase of the Pentium Pro processor to be much more intelligent in terms of predicting program flow. Optimized scheduling requires the fundamental “execute” phase to be replaced by decoupled “dispatch/execute” and “retire” phases. This allows instructions to be started in any order but always be completed in the original program order. The Pentium Pro processor is implemented as 3 independent engines that communicate using an instruction pool:

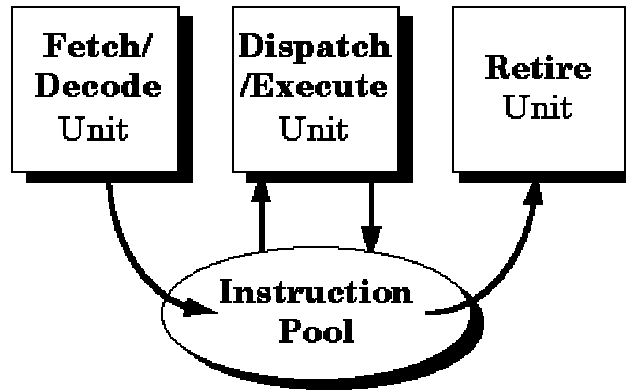


Figure 1. The P6 is implemented as three independent engines that communicate using an instruction pool.

What is the fundamental problem to solve?

Example:

```
r1 ← mem[r0]
r2 ← r1 + r2
r5 ← r5 + 1
r6 ← r6 - r3
```

The 1st instruction causes cache miss. The CPU stalls while waiting for the data → under-utilized. To avoid this memory latency problem the Pentium Pro processor “looks-ahead” into its instruction pool at subsequent instructions and will do useful work rather than be stalled. In the above example, instruction 2 will not be executed since it depends upon the result of instruction 1. However, both instructions 3 and 4 can be executed. Since we must maintain the original program order, the results of instructions 3 and 4 are stored back in the instruction pool awaiting in-order retirement → Instructions are executed out of order.

Dynamic Execution technology can be summarized as optimally adjusting instruction execution by predicting program flow, analyzing the program’s dataflow graph to choose the best order to execute the instructions, then having the ability to speculatively execute instructions in the preferred order.

The Pentium Pro processor pipeline:

- **Fetch/Decode Unit:** An in-order unit that takes as input the user program instruction stream from the instruction cache and decodes them into a series of micro-operations that represent the dataflow of that instruction stream. The program pre-fetch is itself speculative.
- **Dispatch/Execute unit:** An out-of-order unit that accepts the dataflow stream, schedules execution of the micro-operations s.t. data dependencies and resource availability and temporarily stores the results of these speculative executions.
- **Retire unit:** An in-order unit that knows how and when to commit (“retire”) the temporary, speculative results to permanent architectural state.
- **Bus Interface unit:** A partially ordered unit responsible for connecting the 3 internal units to the real world.

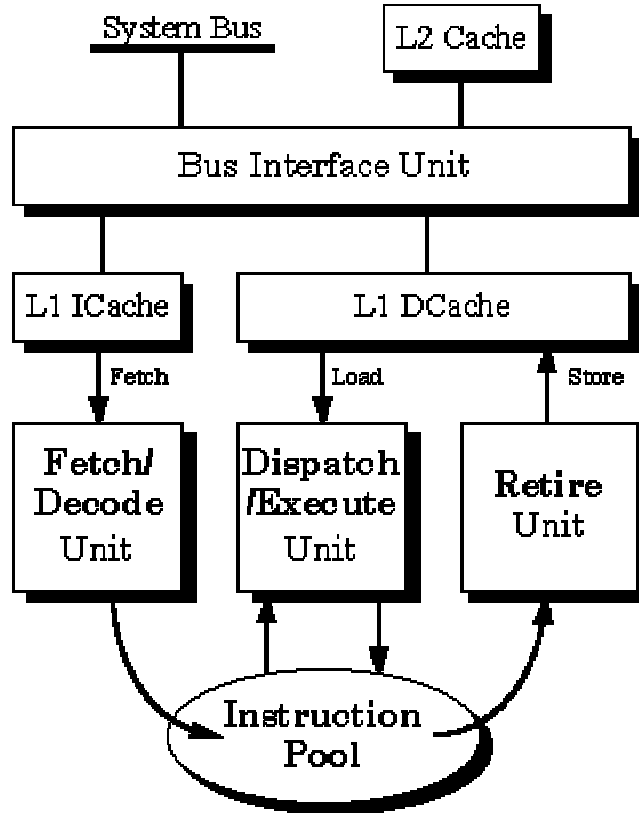


Figure 3. The three core engines interface with the memory subsystem using SR/3K unified caches.

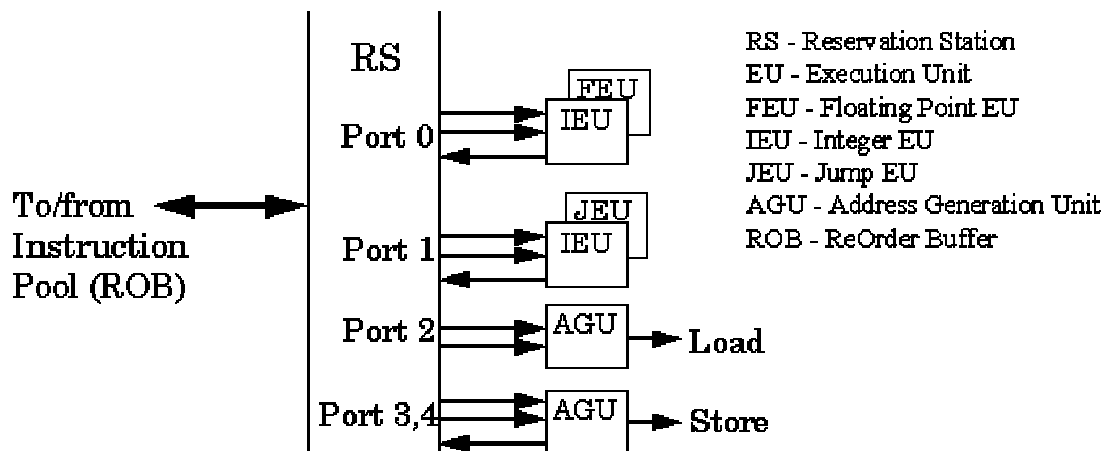


Figure 5: Looking inside the Dispatch/Execute Unit

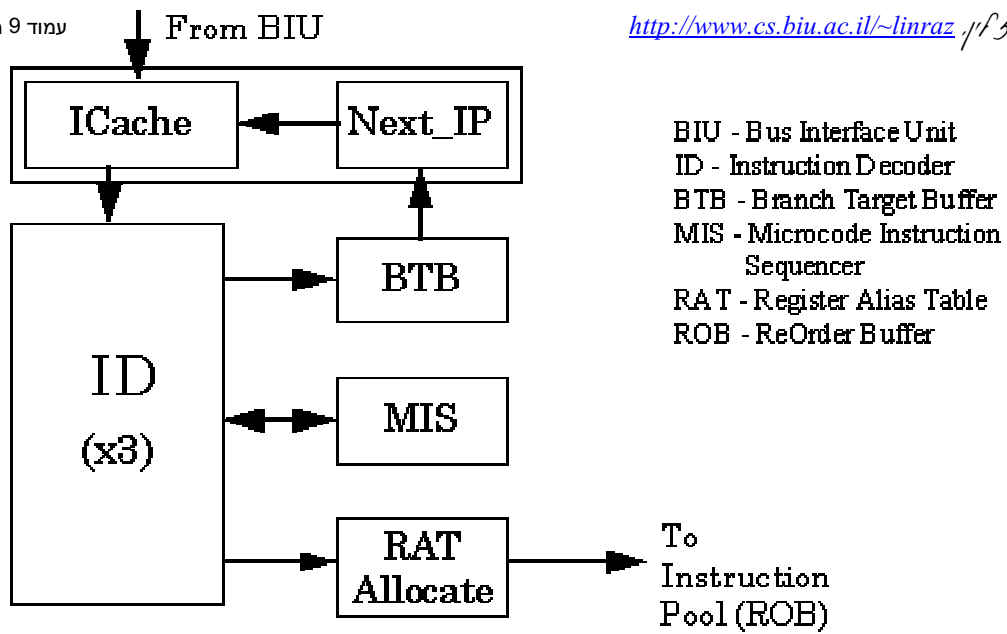


Figure 4: Looking inside the Fetch/Decode Unit

Pentium 4 – שני

Processor Architecture refers to the instruction set, registers and memory-resident data structures.

Processor Micro-architecture refers to implementation of a processor architecture in silicon.

What determines true processor performance?

$$\text{Performance} = \text{MHz} \times \text{IPC}$$

Performance can be improved by increasing frequency, IPC or optimally both. Frequency is a function of both the manufacturing process and the micro-architecture. At a given clock frequency, the IPC is a function of processor micro-architecture and the specific application being executed. Increasing either frequency or IPC and holding the other close to constant with the prior generation can still achieve a significantly higher level of performance.

It is also possible to increase performance by reducing the number of instructions that it takes to execute the specific task being measured (Single Instruction Multiple Data).

Integer and basic office productivity applications tend to have many branches in the code that are difficult to predict, thus reducing overall IPC potential.

Floating point and multimedia applications tend to have branches that are very predictable and thus naturally have a higher average IPC potential.

The Pentium 4 processor uses out-of-order speculative execution and superscalar execution. In Pentium 4 processor a **hyper-pipelined technology (20-stage)** was implemented, where the depth of the pipeline was doubled from that of the P6 micro-architectural generation (Pentium 3). This deeper pipeline delivers significantly higher levels of frequency. The design effort focused on the following:

- **Minimizing the penalty associated with branch mis-predicts:**

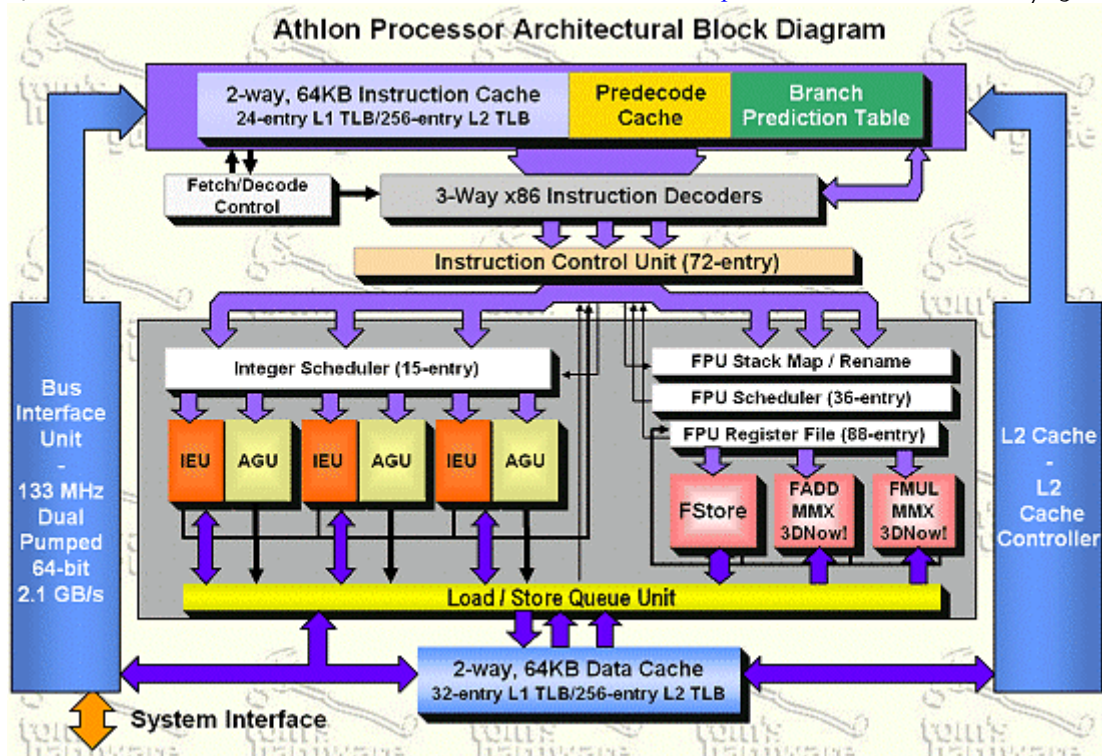
The micro-architecture takes advantage of out-of-order, speculative execution. This is where the processor routinely uses an internal branch prediction algorithm to predict the result of branches in the program code and then speculatively executes instructions down the predicted code branch. If the processor mis-predicts a branch, all the speculatively executed instructions must be flushed from the processor pipeline in order to restart the instruction execution down the correct program branch. On more deeply pipelined designs, more instructions must be flushed from the pipeline, resulting in a longer recovery time from a branch mis-predict → applications that have many, difficult to predict, branches will tend to have a lower average IPC.

- **Keeping the high frequency execution units busy (vs. sitting idle)**

- **Reducing the number of instructions needed to complete a task or program**

Many applications often perform repetitive operations on large sets of data. Further, the data sets involved in these operations tend to be small values that can be represented with a small number of bits. These two observations can be combined to improve application performance by both compactly representing data sets and by implementing instructions that can operate in these compact data sets (Single Instruction Multiple Data) → reducing the overall number of instructions that a program is required to execute.

Using Advanced Dynamic Execution (as in the previous article).



Pentium 3 has 10-stage pipeline, Athlon has 11, and Pentium 4 has 20.

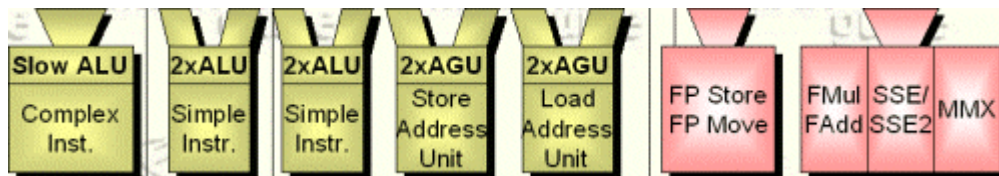
The reason for the longer pipeline is Intel's wish of Pentium 4 to deliver highest clock rates. The smaller or shorter each pipeline stage, the fewer transistors or 'gates' it needs and the faster it is able to run. However, there is also one big disadvantage to long pipelines. As soon as it turns out at the end of the pipeline that the software will branch to an address that was not predicted, the whole pipeline needs to be flushed and refilled. The longer the pipeline the more 'in-flight' instructions will be lost and the longer it takes until the pipeline is filled again.

Pentium 4 pipeline can keep up to 126 instructions 'in-flight', amongst them up to 48 load and 24 store operations. The improved trace cache branch prediction unit described above is supposed to ensure that flushes of this long pipeline are only rare occasions.

The stuff that happens in the trace cache, as mentioned above, only represents the first five stages of the pipeline of Pentium 4. What follows is:

- Allocate resources
 - Register renaming
 - Write into the μ OP queue
 - Write into the schedulers and compute dependencies
 - Dispatch μ OPs to their execution units
 - Read register file (to ensure that the correct ones of the 128 all-purpose register files are used as the register(s) for the actual instruction)
- After that comes the actual execution of the μ OP.

The Rapid Execution Engine:



The basic parts of the 'Rapid Execution Engine' are the two 'double-pumped' ALUs and AGUs. Each of the four is said to be clocked with double the processors clock, because they can receive a μ OP every half clock. Simple μ OPs that can be processed by the Rapid Execution Engine are executed in half a clock, which is obviously a very good thing.

The story looks a lot different for the instructions that cannot be processed by the rapid execution units. Those instructions or μ OPs need to use the one and only 'Slow ALU', which is not 'double pumped'. The majority of instructions need to use this path. However, the majority of code actually consists of

the most simple 'AND', 'OR', 'XOR', 'ADD', instructions, making Intel's 'Rapid Execution Engine'-design sensible though not particularly amazing. Things look worse if you have a look at the red boxes, which represent the FPU-part of Pentium 4.

SSE2 - The New Double Precision Streaming SIMD Extensions

To conclude this epic piece about Pentium 4's internal architecture I need not forget to mention SSE2. 144 new instructions are finally enabling everything that SSE was expected to be in the first place. The 128-bit of packed data, which could only be in form of four single-precision floating-point values under SSE can now be operated in all of the following options:

- 4 single precision FP values (SSE)
- 2 double precision FP values (SSE2)
- 16 byte values (SSE2)
- 8 word values (SSE2)
- 4 double word values (SSE2)
- 2 quad word values (SSE2)
- 1 128-bit integer value (SSE2)

The options are vast and the usefulness undoubted.

Feature	AMD Athlon™ Processor	Pentium® III	Pentium® 4
Operations per clock cycle	9	5	6
Integer pipelines	3	2	4
Floating point pipelines	3	1	2
Full x86 decoders	3	1	1
L1 cache size	128KB	32KB	12k μ op + 8KB Data Cache
L2 cache size	256KB on-chip	256KB on-chip	256KB on-chip
Total on-chip full-speed cache	384KB	288KB	264KB + 12k μ op
Total effective on-chip full-speed cache	384KB (exclusive)	256KB (inclusive)	256KB - 12k μ op (inclusive)
System bus speed	200 MHz to 266MHz	100MHz or 133MHz	400MHz
3D enhancement instructions	Enhanced 3DNow!™	SSE	SSE2
Single-precision FP SIMD	Yes	Yes	Yes
4 FP operations per clock	Yes	Yes	Yes
Cache/prefetch controls	Yes	Yes	Yes
Streaming controls	Yes	Yes	Yes
DSP/comm extensions	Yes	No	Yes
ROB	72	42	126

מאמר שלישי: Crusoe - Transmeta

The Crusoe processor solutions consist of a hardware engine logically surrounded by a software layer. The engine is a very long instruction word (VLIW) CPU capable of executing up to 4 operations in each clock cycle. It has been designed purely for fast low-power implementation using conventional CMOS fabrication. The software layer is called Code Morphing software because it dynamically “morphs” x86 instructions into VLIW instructions. In other words, the Transmeta designers have judiciously rendered some functions in hardware and some in software, according to the product design goals and constraints.

The Transmeta designers have decoupled the x86 instruction set architecture from the underlying processor hardware, which allows this hardware to be very different from a conventional x86 implementation. For the same reason, the underlying hardware can be changed radically without affecting legacy x86 software: each new CPU design only requires a new version of the Code Morphing software to translate x86 instructions to the new CPU’s native instruction set. The Code Morphing software offers opportunities to improve performance without altering the underlying hardware.

Transmeta created a very simple, high performance, VLIW engine with 2 integer units, a floating point unit, a memory (load/store) unit and a branch unit. A Crusoe processor long instruction word, called a *molecule*, can be 64 bits or 128 bits long and contain up to 4 RISC-like instructions, called *atoms*. All atoms within a molecule are executed in parallel, and the molecule format directly determines how atoms get routed to functional units. Molecules are executed in order, so there is no complex out-of-order hardware. To keep the processor running at full speed, molecules are packed as fully as possible with atoms.

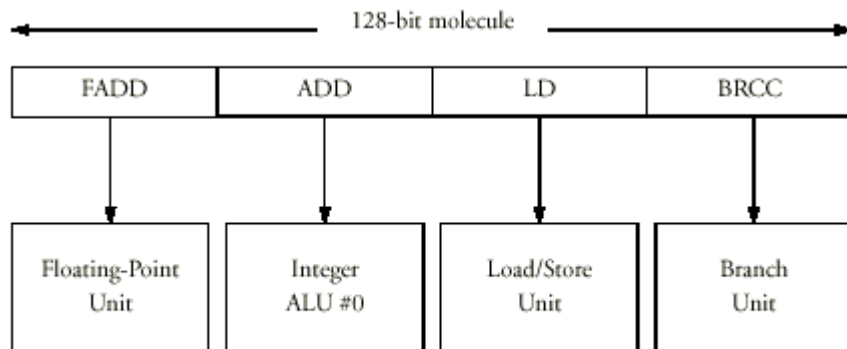


Figure 1. A molecule can contain up to four atoms, which are executed in parallel.

Superscalar out-of-order x86 processors, such as the Pentium 2 and Pentium 3 processors, also have multiple functional units that can execute RISC-like operations (micro-ops) in parallel. The hardware these designs use to translate x86 instructions into micro-ops and schedule (dispatch) the micro-ops to make best use of the functional units:

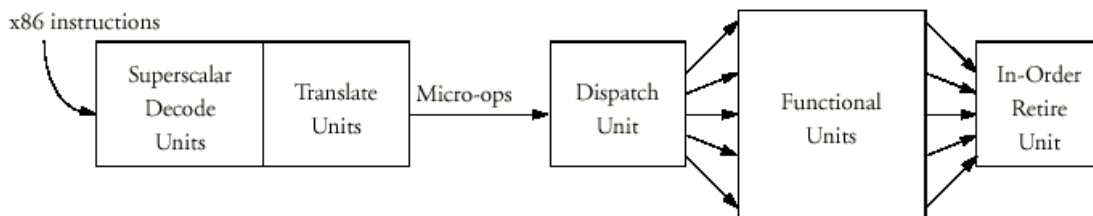


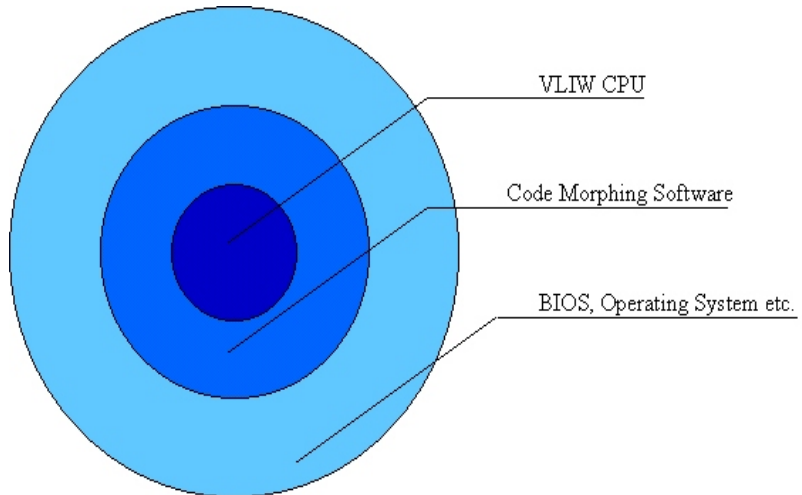
Figure 2. Conventional superscalar out-of-order CPUs use hardware to create and dispatch micro-ops that can execute in parallel.

Since the dispatch unit reorders the micro-ops as required to keep the functional units busy, a separate piece of hardware, the in-order retire unit, is needed to effectively reconstruct the order of the original x86 instructions, and ensure that they take effect in proper order.

Code Morphing Software

The code morphing software's function is to translate instructions from one instruction set architecture (x86 target ISA) into instructions for the VLIW engine that the Crusoe processor uses. The code morphing software is the only piece of software that is written into the Crusoe processors Code Morphing component. The Code Morphing software resides in a 512kb flash chip and this is the first thing to be loaded. The code morphing software can translate an entire batch of x86 instructions at once, the resulting translation is stored into a translation cache. Once translated an instruction does not have to be translate again. The Crusoe translation cache, including the code morphing code, is in a separate memory area that is inaccessible to x86 code. At initialization time the code morphing software copies itself from ROM to DRAM because of faster access times when in DRAM compared to ROM.

As with most caching techniques, the code morphing software takes advantage of "locality of reference." It does this by reusing translated instructions that already reside within its translation cache. When a translation is made the processor does not know how much of the translated code will be of use to it. Most frequently executed code has to be given priority over, say, an instruction that is only executed once. The code morphing software has a number of different modes that it can use to get the best translation



results. These modes include interpretation (slow, but has no translation overhead) through translation with straightforward code generation, all the way to vastly optimized code. This type takes the longest to generate, but runs faster. Feedback from the code morphing software determines the type of translation mode that will be used. Code Morphing Software collects data from previous executions of instructions and uses this history to predict future execution patterns and prepare for them. Code information such as block execution frequencies and branch history is calculated. This data is used to decide what to optimize and translate.

The flexibility of the software translation approach comes at a price: the processor has to dedicate some of its cycles to running the Code Morphing software, cycles that a conventional x86 processor could use to execute application code.

Decoding and Scheduling:

Conventional x86 superscalar processors fetch x86 binary instructions from memory and decode them into micro-operations, which are then reordered by out-of-order dispatch hardware and fed to the functional units for parallel execution.

In contrast, Code Morphing can translate an entire group of x86 instructions at once, creating a **translation**, whereas a superscalar x86 translates single instructions in isolation. Moreover, while a traditional x86 translates each x86 instruction every time it is executed, Transmeta's software translates instructions **once**, saving the resulting translation in a **translation cache**. The next time the (now translated) x86 code is executed, the system skips the translation step and directly executes the existing translated code.

Caching:

The translation cache, along with the Code Morphing code, resides in a separate memory space that is inaccessible to x86 code. The Code Morphing software's technique of reusing translations takes advantage of "locality of reference". As an application executes, Code Morphing "learns" more about the program and improves it so it will execute faster and faster.

Filtering: (idea: code that the program executes very often should be optimized as best as can be)

In typical applications, a very small fraction of the application's code (often less than 10%) accounts for more than 95% of execution time. Therefore, the translation system needs to choose carefully how much effort to spend on translating and optimizing a given piece of x86 code. The Code Morphing software includes in its arsenal a wide choice of execution modes for x86 code, ranging from

interpretation (which has no translation overhead at all, but executes x86 code more slowly), through translation using very simple-minded code generation, all the way to highly optimized code (which takes longest to generate, but which runs fastest once translated).

Prediction and path selection: (*idea:* statistics by branch)

The translator adds code whose sole purpose is to collect information such as block execution frequencies, or branch history. This data can be later used to decide when and what to optimize and translate. For example, if a given conditional x86 branch is highly biased (e.g., usually taken), the system can likewise bias its optimizations to favor the most frequently taken path. Alternatively, for more balanced branches (taken as often as not, for example), the translator can decide to speculatively execute code from both paths and select the correct result later. Analogously, knowing how often a piece of x86 code is executed helps decide how much to try to optimize that code.

Example:

```
A. addl %eax, (%esp) // load data from stack, add to %eax
B. addl %ebx, (%esp) // load data from stack, add to %ebx
C. movl %esi, (%ebp) // load %esi from memory
D. subl %ecx, 5 // subtract 5 from %ecx
```

First: translating the x86 instructions to simple sequence of atoms:

```
A ld %r30, [%esp] // load from stack, into temporary
A add.c %eax, %eax, %r30 // add to %eax, set condition codes.
B ld %r31, [%esp]
B add.c %ebx, %ebx, %r31
C ld %esi, [%ebp]
D sub.c %ecx, %ecx, 5
```

Second: the optimizer applies well-known compiler optimizations to the code, such as: (a) common subexpression elimination, (b) loop invariant removal, or (c) dead code elimination:

```
ld %r30, [%esp] // load from stack only once
add %eax, %eax, %r30
add %ebx, %ebx, %r30 // reuse data loaded earlier
ld %esi, [%ebp]
sub.c %ecx, %ecx, 5
```

Third: the scheduler reorders the remaining atoms and groups them into individual molecules. This process is similar to what out-of-order processors do in their dispatch hardware:

```
1. ld %r30, [%esp]; sub.c %ecx, %ecx, 5
2. ld %esi, [%ebp]; add %eax, %eax, %r30; add %ebx, %ebx, %r30
```

2 important points to observe:

- Though the molecules are executed in-order by the hardware, they perform the work of the original x86 instructions out of order.
- The molecule explicitly encode the instruction level parallelism, hence they can be executed by a simple VLIW engine.

Exceptions and speculation:

In the x86 ISA, exceptions are *precise*: when one instruction causes an exception, all instructions preceding it must complete before the exception is reported and none of the subsequent instructions may complete. In the translation the ordering of the instructions might be different thus we might already execute code and by that to violate the rules of the precise exceptions. Out-of-order processors also have this problem. They employ complex hardware mechanisms to delay or undo the effects of micro-ops that have been executed “too soon”.

The Crusoe solution: all registers holding x86 state are *shadowed*, i.e. there exists 2 copies of each register, a *working* and a *shadow* copy. Normal atoms only update the working copy of the register. When execution reaches the end of a translation without encountering an exception, a special *commit* operation copies all working registers into their corresponding shadow registers. If any x86-level exception occurs inside the translation, the runtime system undoes the effects of all molecules executed since the start of the translation. This is done via a *rollback* operation which copies the shadow register

values back into the working registers. At this point, the Code Morphing software re-executes the x86 instructions conservatively, i.e. in their original program order, to determine the actual location of the exception.

Alias hardware:

When the translator moves a load operation ahead of store operation, it converts the load into a **load-and-protect** and the store into a **store-under-alias-mask**. In the unlikely event that the store operation overwrites the previously loaded data, the processor raises an exception and the runtime system can take corrective action. Using this mechanism it is always safe to reorder memory loads and stores. The alias hardware can also help eliminating redundant load/store atoms.

LongRun power management:

Adjusting the power consumption without turning the processor off by adjusting the clock frequency on the fly. The Code Morphing can also adjust the Crusoe processor's voltage on the fly (since at a lower operating frequency, a lower voltage can be used).

Example: If we need only 200MHz out of the 700MHz we have, then we decrease the clock (instead of 700 to 200) and now, since we're working on a lower clock → we can decrease the voltage ($P = KCV^2$, whereas P = the chip's power [קפסוה], C = clock [MHz], V = voltage)

Note: Crusoe processor is efficient in caching: we divide the code to blocks and put the blocks in the cache. Every time we get to a block already translated, we jump to the code. Since it is done in the software we can take a bigger window to optimize.

Advantages of the Code Morphing software:

The Code Morphing software provides the Crusoe processor with unprecedented flexibility by implementing the complexities of a traditional microprocessor in software. This results in the following advantages over conventional x86 processors:

Traditional x86 Processors	Crusoe Processor with Code Morphing software
Translates single instructions one at time	Translates an entire group of x86 instructions at once
Translates each x86 instruction every time it is encountered	Translates instructions once, saving the resultant translation in a cache for re-use
Full of complex, power-hungry transistors	Much of the processor functionality is implemented in software — less logic transistors, less power

:Understanding the IA-64 Architecture :מאמר רביעי:**Today's architecture challenges:**

- Sequential semantics of the ISA
- Low instruction level parallelism (ILP)
- Unpredictable branches, memory dependencies
- Ever increasing memory latency
- Limited resources (registers, memory addresses)
- Procedure call, loop pipelining overhead.

IA-64 overcomes these challenges:

- Sequentially inherent in traditional architecture.
- Complex hardware needed to (re)extract ILP.
- Limited ILP available within basic blocks.
- Branches make extracting ILP difficult.
- Memory dependencies further limit ILP.

Sequential semantics – The problem:

A program is a sequence of instructions. There's implied order for the execution of the instructions and potential dependence from instruction to instruction. However, high performance needs parallel execution and parallel execution needs *independent* instructions, thus: independent instructions must be (re)discovered.

The compiler knows the available parallelism, but has no “vocabulary” to express it → Hardware must (re)discover parallelism (= complex hardware needed to (re)extract ILP).

In IA-64:

Program is a sequence of *parallel* instructions groups. There's implied order of instructions groups. There's *no* dependence between instructions within the group → independent instructions are explicitly indicated.

The compiler knows the available parallelism, and now *has* the “vocabulary” to express it → Hardware easily exploits the parallelism.

Low instructions level parallelism – The problem:

Branches – frequent; code block – small. Wider machines need more parallel instructions → need to exploit ILP across branches.

Branch unpredictability – The problem:

Branches alter the “sequence” of instructions. ILP must be extracted across branches. Branch prediction has limitations:

- Not perfect, performance penalty when wrong.
- Need to speculatively execute instructions that can fault.
- Need to defer exceptions on speculative operations → more book keeping overhead hardware.

In IA-64:

Prediction is done by transferring control flow to data flow → prediction removes/reduces branches and enables and enhances ILP.

Unpredictable branches are removed → misprediction penalties are eliminated.

ILP within the basic block increases → both “then” and “else” are executed in parallel.

Wider machines are better utilized.

Each instruction contains three 7-bit GPR register fields (128 integer + 128 floating point registers).

Register set (integer vs. floating point) is determined by instruction.

IA-64 compilers will have to be even smarter than RISC compilers:

- Must attach predicates to conditional branches to aid in speculative execution.
- Must analyze potential parallelism to set template bits appropriately.
- Must check for loads from memory and insert speculative load instruction earlier in stream and replace normal load with speculative check instruction.

IA-64 instructions are fixed length – about 40 bits long.

Instructions can be out-of-order and they can originate from different paths of a branch.

Branch prediction enables chip to execute parallel branches speculatively and discard unneeded results.

Memory dependencies – The problem:

Loads are usually at the top of a chain of instructions. ILP extraction requires moving these loads.

Memory latency – The problem:

Need to distance loads from their uses.

Resource constraints – The problem:

1. Small register space → limits compilers ability to “express” parallelism and creates false dependencies (can be overcome by renaming).
2. Shared resources: condition flags, control registers and etc. It forces dependencies on otherwise independent instructions.
3. Floating point resources.

Loop optimization overhead – The problem:

Loops are a common source of good ILP. Unrolling/Pipelining exploit this ILP. Prologue/Epilogue cause code expansion. Unrolling causes more code expansion and thus limits the applicability of these techniques.

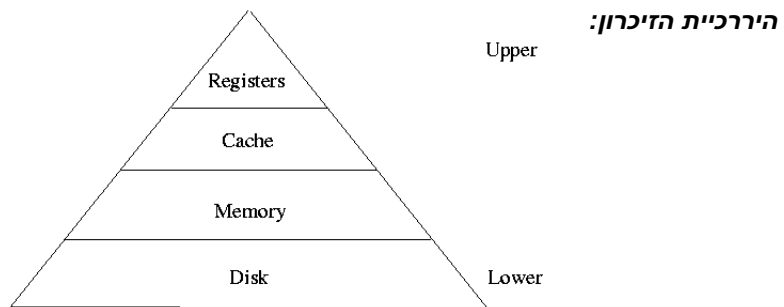
מערכת הזיכרון:

CPU יכול להיות מהיר ככל שהזיכרון יאפשר. כדי שנוכל לקיים את הבקשות בלי השהיות ה-bandwidth צריך להיות:

$$\text{memory reference/instructions} \times \text{bytes per reference} \times \text{IPC/Clock cycle}$$

↓
 כמה פעמים ניגשים לזיכרון עבור פקודה
 כמה בתים שולפים כל פעם שניגשים לזיכרון

מקומיות בזמן (temporal locality): אם השתמשנו ביחידת נתונים לאחרונה ← יש סיכוי גבוה שנשתמש בה שוב בקרוב.
מקומיות במרחב (spatial locality): אם השתמשנו ביחידת נתונים לאחרונה ← יש סיכוי גבוה שנשתמש בנתונים שכנים (למשל: במערכים).



Cache: שיפור הביצועים ע"י מימוש מקומיות זמנית (שומרים בלוקים שניגשנו אליהם לאחרונה) ומקומיות מרחבית (חלוקת הזיכרון לבלוקים). ניהול ה-cache מבוצע בחומרה. מחלקים את הזיכרון לבלוקים של 16 או 32 בתים. בתוך ה-cache שמים יחידות של בלוקים מלאים (יש alignment של הרוחב). את הבלוקים שמים ב-block frame שבו יש state, address tag and data. את ה-address tag מחלקים ל-2:
 1. tag – מאפיין את ה-block.
 2. offset – המיקום בתוך ה-block.

- בעת גישה לזיכרון בודקים: אם ה-tag שצריך נמצא כבר – hit. אחרת – miss ואז:
1. מחליפים בלוק ישן.
 2. מקבלים את הבלוק החדש מהזיכרון.
 3. שמים את הבלוק ב-cache.
 4. מחזירים את המילה הנחוצה מהבלוק.

דוגמה: יש לנו cache block frame בגודל 16 בתים.

lw \$4, 0x128
 בודקים אם 0x120 tag נמצא ב-cache (0x128 mod 16 = 0x128 & 0xffffffff). אם לא – אזי מקבלים את ה-block:

state	tag	data
valid	0x120	0xffffffff, 0x1, 0x7, 0x3

התו הימני ביותר (8) הוא ה-offset ולכן מחזירים ל-CPU את 0x7 ומאחסנים אותו ב-\$4.

lw \$5, 0x124
 בודקים אם 0x120 tag נמצא ב-cache? כן, ולכן מחזירים ישיר את 0x1 ל-CPU.

$$\text{mean access} = \text{cache access} + \text{miss ratio} * \text{main memory access}$$

דוגמה: אם לקיחת נתון מה-cache דורש cycle אחד. לעומת זאת בלי cache זה יעבוד פי 50 יותר לאט (50 cycles) ובאחוז אחד מהמקרים צריך לגשת לזיכרון אזי: $\text{mean access} = 1 + 0.01 * 50 = 1.5$

הגדרות:

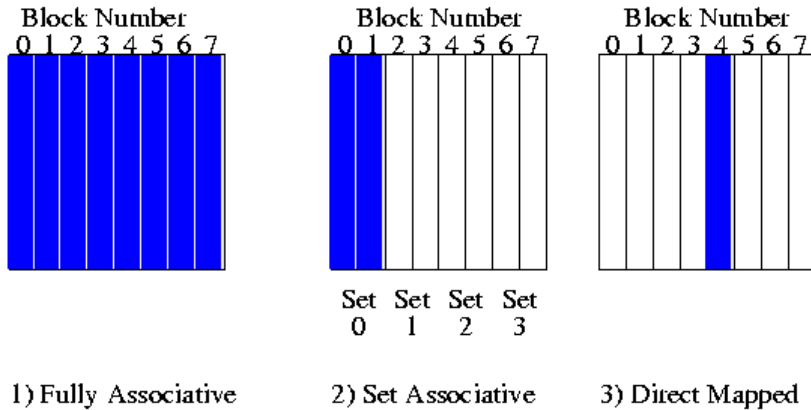
- block – היחידה המינימלית שניתן לייצג.
- hit – הבלוק נמצא ב-cache.
- miss – הבלוק לא נמצא ב-cache (working set > cache size).
- miss ratio – אחוז הגישות שגורמות ל-miss.

hit time – זמן גישה ל-cache.
miss penalty – הזמן שלוקח להחליף בלוק ב-cache + הבאת הבלוק ל-CPU.

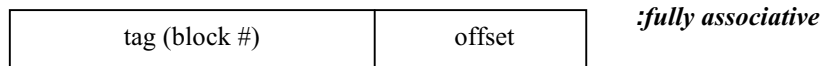
Block Placement

- **fully associative**: הבלוק יכול להיות בכל frame. יתרון: יעיל יותר. אם אני קורא a ו-b שהאינדקס שלהם זהה, הם תמיד יכולים להישאר ב-cache בלי להפריע אחד לשני. חסרון: כל פעם צריך לחפש במקביל.
- **direct mapped**: הבלוק נכנס ל-frame אחד בלבד: אם אני קורא a ו-b שהאינדקס שלהם זהה, אזי אני כל הזמן עושה החלפה ביניהם.
- **set associative**: הבלוק נכנס לקבוצה אחת בלבד.

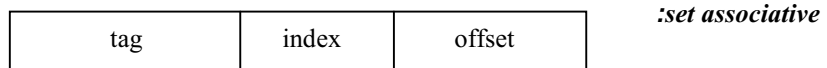
דוגמה: לאן ילך בלוק 12 (1100)?



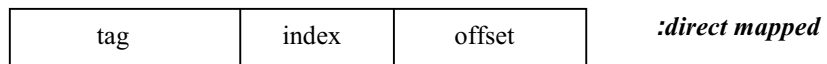
1) Fully Associative 2) Set Associative 3) Direct Mapped



0xa12 → tag = block # = a1, offset = 2.



כל אינדקס יכול להיות רק בקבוצה אחת. יש כאן 4 קבוצות ולכן גודל האינדקס יהיה 2.

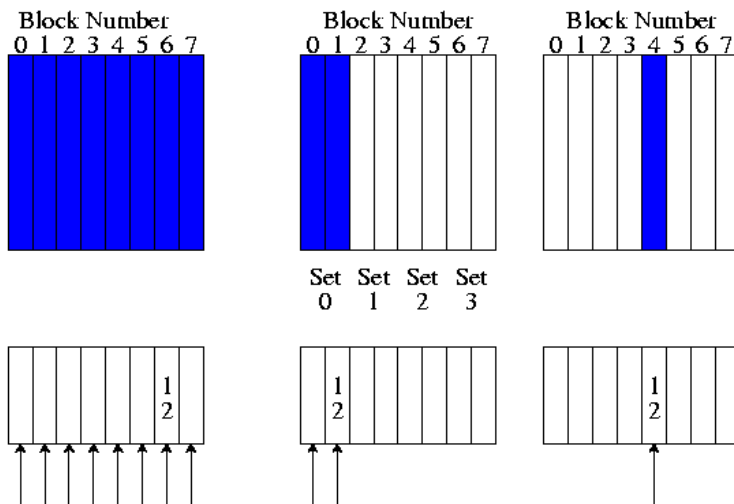


כל cache line מגיע למקום ספציפי. ה-index בוחר את המקום בתוך ה-cache. בדוגמה הנ"ל האינדקס יהיה בעל 3 ביטים וה-tag והאינדקס מהווים את ה-block #.
 0xa12 → a1 = 10100 001 → tag = block # = a1 = 20, offset = 2, index = 1

איך מוצאים בלוק ב-cache?

- השוואת ה-tag.
- חיפוש מקבילי למציאת lookup.
- בדיקת ה-valid bit.

דוגמה: איך נמצא את בלוק 12?



1) Fully Associative 2) Set Associative 3) Direct Mapped

ב-fully associative הבלוק יכול להימצא בכל כתובת ואנחנו רוצים למצוא אותו בפעימת שעות אחת ← צריך יותר חומרה (כדי לאפשר בדיקה במקביל) ← צריך זיכרון אסוציאטיבי.

ב-direct mapped מסתכלים על 3 הביטים הנמוכים ביותר של 12 ומקבלים 4 (12 = 1 100).

איזה בלוק מחליפים כתוצאה מ-miss?

1. LRU – אופטימלי למקומיות זמנית.
2. רנדומלי – כמעט טוב כמו LRU. פשוט יותר.
3. NMRU – Not Most Recently Used – עוקבים אחרי ה-most recently used ובוחרים באופן רנדומי מהאחרים.
4. **אופטימלי** (Belady's Algorithm) – מחליפים את הבלוק שבו נשתמש עוד פעם הכי מאוחר (offline).

סוגי ה-miss:

1. *compulsory (miss in infinite cache)*: בגישה הראשונה לבלוק. זהה לכל סוגי ה-cache.
2. *capacity (miss in fully associative cache)*: קורה כאשר ה-cache אינו מספיק גדול.
3. *conflict*: קורה בגלל שיטת המיפוי, למשל: עבור cache שאינו fully associative יכול להיות שיש 2 מקומות ב-working set שמגיעים לאותו אינדקס. הגדלת אסוצ' ← הקטנת conflict misses.
4. *coherence*: קורה עקב invalidations ממעבדים אחרים (כאשר יש זיכרון משותף לכמה מעבדים). **פתרון**: *memory snooping*: הזיכרון הוא על bus משותף. ברגע שנכתב ערך לכתובת שנמצאת ב-cache של המחשב השני, המחשב השני מאזין לפס, רואה את זה ולכן בגישה הבאה שלו לכתובת הזו הוא יבטל את המיקום ב-cache ויביא אותו חזרה מהזיכרון. ה-snooping טוב ל-write through בלבד.

גודל ה-cache:

גודל ה-cache הוא גודל אחסון הנתונים (לא כולל ה-tag) של ה-cache:

- גודל גדול יותר יכול לנצל מקומיות זמן טוב יותר.
- גודל גדול יותר זה לא תמיד טוב יותר.

Cache גדול מדי ← איטי יותר; זמן גישה עשוי להיות גרוע במסלולים קריטיים.
Cache קטן מדי ← לא מנצל מקומיות זמן טוב; מידע חשוב מוחלף יותר מדי מוקדם.

block size: גודל קטן מדי ← לא מנצלים טוב מקומיות מרחבית; יש tag overhead.
גודל גדול מדי ← יש העברות מידע לא נחוצות; מידע נחוץ מוחלף מוקדם מדי עקב זה שיש מספר כולל של בלוקים קטן.

Associativity: larger associativity: פחות miss rate.
smaller associativity: עלות נמוכה יותר; בד"כ hit time מהיר.

ב-set associative cache לעומת direct mapped cache יש פחות miss ratio אבל זמן הגישה האפקטיבי גרוע יותר.

Write Policies: קריאות נעשות במקביל עם השוואת ה-tag. אולם כתיבות לא נעשות כך ולכן הכתיבות איטיות יותר.

כאשר יש hit, האם מעדכנים את הזיכרון?
כן ← write-through (store-through).
לא ← write-back (store-in, copy-back).

כאשר יש misses, מקצים בלוק ב-cache?
כן ← write-allocate (usually with write-back).
לא ← no-write-allocate (usually with write-through).

Write back

- עדכון הזיכרון רק כשמחליפים בלוק.
 - משתמשים ב-dirty bit, כך שבלוקים נקיים מוחלפים ללא עדכון מהזיכרון.
 - $traffic/reference = f_{dirty} \times miss \times B$
 - יש פחות traffic ל-caches גדולים.
- write back יעיל יותר מ-write through שכן כותבים ל-cache בלבד.

Write through

- עדכון הזיכרון בכל כתיבה.
 - שמירת הזיכרון מעודכן.
 - $traffic/reference = f_{writes}$
 - אין תלות ב-performance של ה-cache.
- לפתרון בעיות coherence של הזיכרון כותבים לזיכרון. ה-cache יותר אפקטיבי לקריאות ולא לכתיבות. הכתיבה לזיכרון לוקחת הרבה cycles.