Zweierkomplement: invertieren + 1

Prozessor-Ablauf

Prozessor... 1. ...fordert Wert von Adresse an, beim Befehlszeiger. 2. ...decodiert Instruktion aus Wert. 3. ...wählt den zur Instruktion gehörenden Baustein aus. **Aktiver Baustein...** 4. ...decodiert Parameter aus Wert. 5. ...liest aus den Registern. 6. ...führt Berechnung aus. 7. ...schreibt in die Register. 8. **Prozessor** erhöht Befehlszeiger entsprechend der Länge der Instruktion.

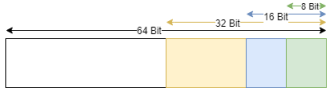
Byte-Order

32 Bit 87654321:
Little Endian: 21 43 65 87
Big Endian: 87 65 43 21

Byte 1 Byte db 0x35
Word 2 Byte, dw 0x2135 ↔ db 0x35, 0x21
Doubleword 4 Byte dd 0x2135 ↔ db 0x35, 0x21, 0x00, 0x00
Quadword 8 Byte dq
Double Quadword 16 Byte

Register

instruction pointer: ip in 16-bit, eip in 32-bit, rip in 64-bit



1	RAX	EAX	*AX*	AL
2	RBX	EBX	*BX*	BL
3	RCX	ECX	*CX*	CL
4	RDX	EDX	*DX*	DL
5	RSI	ESI	SI	SIL
6	RDI	EDI	DI	DIL
7	RSP	ESP	SP	SPL
8	RBP	EBP	BP	BPL
9	R8-R15	R8D-R15D	R8W-R15W	R8L-R15L

1. Accumulator, 2. Datenpointer, 3. Counter für Schleifen, Stringoperationen, 4. Pointer für I/O-Operationen, 5. Quelindizes für Stringoperationen, 6. Zielindizes für Stringoperationen, Exitcode, 7. Stackpointer, Adresse des allozierten Stacks, 8. Basepointer, Adresse innerhalb des Stacks, Basis des Rahmens der Funktion , 9. Zusätzliche Register

Assembler

mov Ziel, Quelle
global x, y; extern z; w

0010	l	.text	0000	w
0000		*UND*	0000	z
0018	g	.text	0000	x
0020	g	.text	0000	y

1.Spalte (Offset), die letzte den Namen
2.Spalte (Symbolattribute: g für global und l für lokal)
3.Spalte (Sektion (für unsere Fälle .text))
extern deklarierte Label z: *UND*
Deklaration Label

O-Datei	Assem.	C	Dek. C
lokal	–	global, inter. Linkage	static
global	global	global, exter. Linkage	–
–	extern	extern	extern

extern = diesem oder anderen File deklariert
lokal = nicht rausgeben
global = anderen o-files zur verfügung stellen

Adressierungsmodi

(Displacement): mov rax, [0x1000]
(Base): mov rax, [rcx]
(Index * Scale): mov rax, [rbx * 4]
Label nicht übersetzt, Offset des nachfolgenden Befehls assoziiert

Arrays

Startadresse, Offset = Index · Variablengrösse in Byte
add rax, [a + 0 * 8]

Arithmetische und logische Operatoren

`add z, q`	$z \leftarrow z + q$
`sub z, q`	$z \leftarrow z - q$
`adc z, q`	$z \leftarrow z + q + c$
`sbb z, q`	$z \leftarrow z - q - c$
`neg z`	$z \leftarrow 0 - z$
`inc z`	$z \leftarrow z + 1$
`dec z`	$z \leftarrow z - 1$

Bits	Res. MSBs	Res. LSBs & 2.Operand
64	RDX	RAX
32	EDX	EAX
16	DX	AX
8	AH	AL

mul rbx, dass RDX:RAX ← RAX · RBX
imul Signed verwendet nur LSB
div und idiv wie mul; Quotienten in LSB, Rest in MSB

Links-Shift — Multiplikation einer Binärzahl mit 2^m
1111 Linksshift um 2^1 = um 2^m
11110

Rechts-Shift — Division einer Binärzahl
1111 um 2^1 = 0111

Sign-Extension (Arithmetischer Rechts Shift)

Statt 0 wird links das Vorzeichen reinkopiert

Relative Sprünge

JMP zahl = RIP←RIP + zahl
JMP label = RIP←label

Flags

gemeinsames Register RFLAGS
Carry Flag - CF Überlauf unsigned
Overflow Flag - OF Überlauf signedwerden immer beide bestimmt
Zero Flag - ZF Resultat = 0
Sign Flag - SF MSB des Resultats
Parity Flag - PF niederwertigste Byte = gerade Anzahl 1

Condition Codes

CC	Name	Flags
A	Above	CF = 0 und ZF = 0
AE	Above or Equal	CF = 0
B	Below	CF = 1
BE	Below or Equal	CF = 1 oder ZF = 1
E	Equal	ZF = 1
G	Greater	ZF = 0 and SF = OF
GE	Greater or Equal	SF = OF
L	Less	SF ≠ OF
LE	Less or Equal	ZF = 1 und SF ≠ OF
PE	Parity Even	PF = 1
PO	Parity Odd	PF = 0

verwirft Ergebnis, setzt Flags:
cmp: cmp rax, rbx berechnet RAX – RBX
test: test rax, rbx berechnet RAX & RBX

Bedingte Anweisungen

CMOVcc: Conditional MOV, Jcc: Conditional JMP, SETcc: Schreibt 1 ins 8-Bit grosse Ziel, wenn CC erfüllt, sonst 0

Stack

Calling Convention Vereinbarung zw. Caller & Callee. Wird durch Betriebssystem bestimmt: Argumente, Rückgabewerte, Register bewahren, wer setzt Stackframe?
Prolog Funktion: push rbp; mov rbp, rsp
Epilog Funktion: mov rsp, rbp; pop rbp
call x legt die nächste Adresse auf den Stack und springt zu x
ret nimmt die Rucksprungadresse vom Stack und springt dahin
sub rsp , 0x20 ; allocates 0x20 Bytes on stack
add rsp , 0x20 ; deallocates 0x20 Bytes on stack

C

1. Präprozessor
2. Compiler
3. Assembler .asm → Objektdatei/Binärsequenz .o
4. Linker mehrere .o-Dateien → Executable

Objekt-Datei: Enthält Binärsequenzen

Executable: Jedes Symbol erhält einen eigenen, festen Platz im Executable

Präprozessor

1. Durchlauf
Entfernen aller Kommentare, fortgesetzte Zeile → in eine Zeile
2. Durchlauf, Tokenization
Bezeichner: ~~Sonderzeichen~~, zusammenfassen: "", escapen
3. Durchlauf
Präprozessor-Direktiven (#) ausführen, Makros durch Expansion ersetzen
<file.h> Systemfolder, "file.h" aktuelles Verzeichnis,
Header 1 / Header 2 → Präpozessor → Translation Unit → Compiler

Variablen

globale Variable gleich wie in Assembler, Speicher wird fix reserviert, Grösse durch Typ bestimmt
Variable = Label auf Assembler-Ebene
Intialisierung: `int x = 15;` ↔ x : dd 15

Objekt

zusammenhängender Speicherbereich, Inhalt kann als Wert interpretiert werden **Objektgrösse bestimmen:** `sizeof(T)`

Basistypen

`char` 8 Bit, `short` 16 Bit, `int` 16 Bit, `long` 32 Bit, `long long` 64 Bit
default-mässig signed, unsigned muss sonst strikt angegeben werden
const beezieht sich auf den links ausser wen ganz links dann rechts
Mögliche Konvertierungen sizeof(x) vom Stack: %i = int, %X = int(Hexzahl), %p = void *, %s = char *

Arrays

Bezeichner eines Arrays → Pointervariable
Mit Elementtyp T ist `a[index]` äquivalent zu `a + sizeof(T) * index` `f(int x[], int len)` = `f(int* x, int len)`
`size_t` zum iterieren
```
size_t n = sizeof a / sizeof a [0];
// b == 5 ( elements )
```
String iterieren: `while (pc != '\0')`

Structs

```
struct T
{int x; int y;};
struct T t, u;
```

belegt gleichen Speicherplatz wie `int x; int y;` einzeln
Zugriff: `t.x = t.y`
x gleiche Adresse wie Struct
Member müssen im Speicher nicht dicht liegen (Padding möglich)

Cache

Arbeitsbereich sind alle Speicherzellen, die in einem Zeitraum verwendet wurden

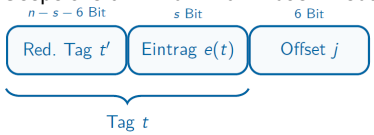
Fully Associative Cache (FAC)

Eintrag i besteht jeweils aus Adresse ai und Datenbyte di = [ai]:
Zu jeder Zeile ein Hardware-Baustein, der den Tag t überprüft Überprüfung wird parallel gemacht

+ Lokalitätsprinzip bestmöglich ausgenutzt
- Lookups benötigen viel Hardware und sind teuer

Direct-Mapped Cache (DMC)

Anzahl Einträge/Zeilen = 2^s
eine Cachezeile nur an einem einzigen Ort möglich
Index des Eintrags wird durch die untersten s Bit des Tags bestimmt
Gespeichert wird nur das reduzierte Tag

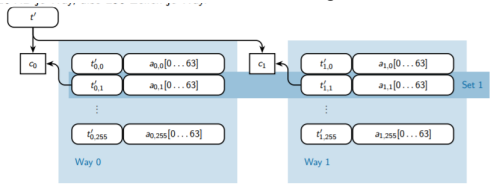


Lookup

ein einziger Vergleichsbaustein
Vorgehen: 1. hinteren 6 Byte abschneiden 2. Hintere s bits nehmen und an die Index-Stelle gehen 3. Rest der Adresse mit Reduziertem Tag vergleichen.
+ einfach zu implementieren + sehr schneller Lookup - viele Kollisionen (1234_h, $AB34_h$ bei s=8 gleicher Eintrag)

Set-Associative Cache (SAC)

parallele Verwendung von k Direct-Mapped Caches
Jede Cachezeile kann in k Einträgen gespeichert werden
Anzahl Set = k
Jeder DMC = WAY
Set-Nummer = Nummer des Eintrags i



weniger komplex als FAC, weiniger Kollisionen als DMC, Genau so schnell wie FAC und DMC

Random Access Memory (RAM)

Dynamischer Speicher (Heap)

Nur explizite Speicherfreigabe durch OS vorgesehen:
Reservieren: `void * malloc (unsigned int s)`
Reserviert Speicherblock der Grösse s
gibt Adresse des allozierten Speicherblocks zurück
freigeben: `void free (void* p)`

Interne Fragmentierung: grösserer Speicherblock als benötigt **Externe Fragmentierung:** Programm reserviert immer wieder Speicher, gibt unregelmässig frei

Feste Blockgrösse

Dezentrale Speicherung: Überläufe **Zentrale Speicherung:** Speicherplatz muss extra reserviert werden **Bitlisten:** 0 Block ist frei, 1 verwendet **Verkettete Listen:** Status(frei?), Start(Adresse erster Block), Size(Anzahl Blöcke), Next(Pointer nächstes Listenelement)

Bei Freigabe wird geprüft ob vordere/hintere Teil ebenfalls frei ist und bei Möglichkeit verschmolzen.

Suchalgorithmen

First Fit: Wählt erste passende Lücke am Anfang **Next Fit:** Wählt erste passende Lücke nach zuletzt reserviertem Bereich **Best Fit:** Durchsucht alle Lücken und wählt die kleinste passende aus **Worst Fit:** Durchsucht alle Lücken und wählt die grösste aus

Grössenklassen

Bereiche nur in bestimmten Grössen, freie Bereiche in Liste, **Quickfit:** wählt kleinstpassenden aus Liste +Schnelle Reservation, -Nachbarn sind nicht leicht zu finden

Buddy-System

Wenn 2 Bereiche gleiche bits bis auf einen → Buddies

Objekt-Pools

Speicherbereich fester Grösse(Page) in kleinere Bereiche, keine Rekombi bei Rückgabe, Mehr Objekte?→neue Page, Freie Objekte in Freiliste

Programmstart/-ende

Seperates Register für Syscalls: IA32_LSTAR Intel Prozessoren haben Privilege-Levels: 0 = OS, 3 = Programme
mov rax, [Funktionscode]; mov rdi, [weitere Argumente]; syscall;

Virtueller Speicher (Hardware)

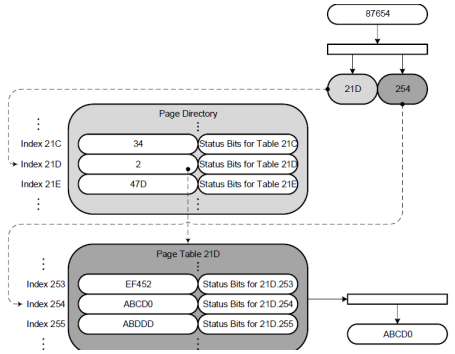
MMU (Memory Management Unit) übersetzt Virtuelle in reale Adresse
Pro Prozess eine Page-Table
OS stellt Übersetzungs-Mapping bereit. Prozesse bekommen keine realen Adressen mit
Bei falschem Zugriff → Fault-Interrupt → OS übernimmt
OS lagert nicht benötigtes aus, OS lädt fehlende inhalte vom Sekundärspeicher Pages normalerwiese 4KB gross
Hauptspeicher besteht aus Pageframes
Wenn Pages (4kb/12bit) Hauptspeicher-Adresse = 0x AB 78 90 00 → Page Frame Number = 0xAB789
Page = Daten

Single-Level Page-Table

ein Eintrag pro Page + Lookup sehr schnell → Index = PageNumber

Two-Level Page-Table

Page Number wird aufgeteilt: Directory Index (nur eine!), Page Table Index (zweidimensionales Array)

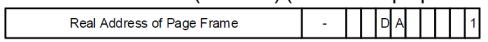


Multi-Level Page-Table

Page Directory Pointer Table, Page Directory, Page Table **Translation Lookaside Buffer:** Cache für häufig benu. Mappings

Virtueller Speicher (Software)

Unterstes Bit = P-bit (Present) (1 = im Hauptspeicher)



MMU setzt A-Bit/«Accessed» bei jedem Zugriff auf Page, D-Bit/«Dirty» bei jedem Schreibzugriff auf Page→zurückschreiben in HDD
OS kann beide bits löschen
Wenn P-bit = 0 → Der Platz davor wird dafür verwendet, wo die Page im Sekundärspeicher liegt
Wenn Page nicht alloziert ist → Schutzverletzung
Nachdem Page geladen wurde wiederholt MMU den Zugriff

Dreschen/Häufiges Pagen

Hauptspeicher viel zu klein/zu viele Prozesse→mehr Hauptspeicher, Beschränkung Anz. Prozesse, Verminderung Paging-Strategien

Ladestrategien RAM←HDD

Beeinflusst Häufigkeit von Pagefaults
Demand Paging: Laden auf Anfrage, +min. Aufwand, -lange Wartezeiten **Prepaging:** Seiten frühzeitig geladen **Demand Paging mit Prepaging:** wie Demand + benachbarte Pages(Lokalitätpr.), +weniger Page-Faults, +Blocktransfer, -Manchmal unnötiges Laden

Entladestrategien RAM→HDD

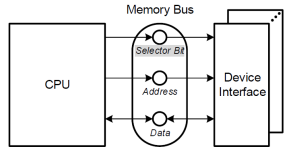
Beeinflusst Zeit bei Pagefaults
Demand Cleaning: nur geschrieben wenn nötig, +min. Aufwand, -erhöhte Wartezeit **Precleaning:** Vorausschauendes Schreiben, +reduzierte Wartezeit, -wenn Pages nach Schreibvorgang nochmals geändert **Page Buffering:** 2 Listen: **U**nmodified Pages (zuerst ersetzt), **M**odified Pages
OS schreibt von M in Secondärspeicher/ OS schiebt von U zu M wenn Dirtybit gesetzt. +/-Precleaning, +schnelle Auswahl beim Ersetzen

Verdrängungsstrategien

Beladys Anomalie: Grösserer Hauptspeicher kann zu mehr Page Faults führen **Optimal:** spätesten in Zukunft gebraucht **FIFO:** Problem: alte, häufig benutzte Pages werden gelöscht und gleich wieder geladen **Second Chance:** FIFO + prüft Referenced-Bit, 0=weg, 1=hinten + 0 **Clock:** Effiziente Impl. Second Chance: Linked List im Kreis. Pointer wird nur verschoben **Least Recently Used:** ersetzt längste unbenutzte Page, notiert Zeitpunkt in Page-Table, + sehr nahe am Optimum - grosser Aufwand in HW, Page-Einträge grösser **Working Set:** Pro Page-Interrupt
R==1: t = now setzen,R==0: now - t<Working Set T→JA: Page behalten / NEIN: Page entfernen

In-/Output

Memmory-mapped I/O: pro Gerät 1 Adressbereich, -Adressbereich kann nicht für Speicher benutzt werden, +Einfachheit **Port-mapped I/O:** separater Bus, zwei Adressräume (Speicher und Geräte), Pro Adressraum eigene Instruktionen, -Komplexität **Port-mapped I/O via Memorybus:** gemeinsamer Bus, zusätzliche Bitleitung für Speicher|I/O, Adressraum wird um 1 Bit erweitert, +Ganzer Adressraum, +einfaches Design



Kommunikationsmechanismen

Programmgesteuert/Polling Programm fragt die ganze Zeit ab, +Keine Verzögerung, -Legt CPU lahm **Polling ohne Busy-wait** Programm pollt in Abständen, -erfordert zeitliche analyse, +CPU kann etwas anderes tun **Interruptgesteuert**: Device meldet sich wenn Ready, Prozessor hat Interrupt Pin + Interrupt Number. Prozessor hat Interrupt Vector Table(Tabelle mit Funktionen), welche der Prozessor anhand der Interrupt Number aufruft Nach Instruktion prüfen ob interupt Pin gesetzt ist. Ja → unterrbricht Ausführung & sichert alles

Treiber

ohne Treiber → -Sicherheit, -Stabilität, -Komplexität, -Multiprogrammierung **Mit Treiber** Benutzerprogramme können nur über OS-API auf Hardware zugreifen
Treiber = Komponente zur Kommunikation mit HW / Separat ins OS integrierbar / können aufeinander aufbauen

Varia

Nibble = 4 Bit, Oktett = 8 Bit, Byte = Oktett **unsigned/ohne Vorzeichen**: $0...2^n - 1$
Disjunktion: $\vee$, **Konjunktion:** $\wedge$
2^{30} G Giga
2^{40} T Tera
2^{50} P Peta
2^{60} E Exa