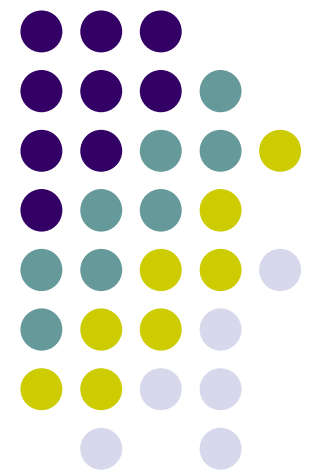
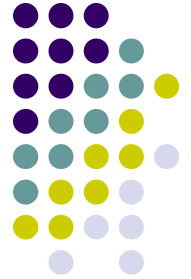


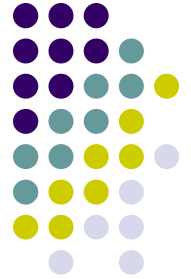
Computer Architecture

Assoc. Prof. Nguyễn Trí Thành, PhD
UNIVERSITY OF ENGINEERING AND TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS
ntthanh@vnu.edu.vn





Intel-based Assembly



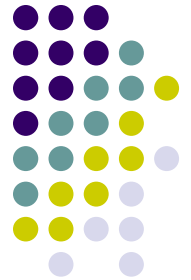
Intel-based Assembly

INTEL HISTORY

Intel Microprocessor History



Processor History



1 The 4-bit processors

1.1 Intel 4004

1.2 Intel 4040

2 The 8-bit processors

2.1 8008

2.2 8080

2.3 8085

3 Microcontrollers

3.1 Intel 8048

3.2 Intel 8051

3.3 Intel 80151

3.4 Intel 80251

3.5 MCS-96 Family

4 The bit-slice processor

4.1 3000 Family

5 The 16-bit processors: MCS-86 family

5.1 8086

5.2 8088

5.3 80186

5.4 80188

5.5 80286

tory



- 1 The 4-bit
 - 1.1 Int
 - 1.2 Int
- 2 The 8-bit
 - 2.1 80
 - 2.2 80
 - 2.3 80
- 3 Microcon
 - 3.1 Int
 - 3.2 Int
 - 3.3 Int
 - 3.4 Int
 - 3.5 MC
- 4 The bit-s
 - 4.1 30
- 5 The 16-b
 - 5.1 80
 - 5.2 80
 - 5.3 80
 - 5.4 80
 - 5.5 80
- 6 32-bit processors: the non-x86 microprocessors
 - 6.1 iAPX 432
 - 6.2 i960 aka 80960
 - 6.3 i860 aka 80860
 - 6.4 XScale
- 7 32-bit processors: the 80386 range
 - 7.1 80386DX
 - 7.2 80386SX
 - 7.3 80376
 - 7.4 80386SL
 - 7.5 80386EX
- 8 32-bit processors: the 80486 range
 - 8.1 80486DX
 - 8.2 80486SX
 - 8.3 80486DX2
 - 8.4 80486SL
 - 8.5 80486DX4
- 9 32-bit processors: P5 microarchitecture
 - 9.1 Original Pentium
 - 9.2 Pentium with MMX Technology

1 The 4-bit	6 32-bit proc	10 32-bit processors: P6/Pentium M microarchitecture
1.1 Int	6.1 iAPX	10.1 Pentium Pro
1.2 Int	6.2 i960	10.2 Pentium II
2 The 8-bit	6.3 i860	10.3 Celeron (Pentium II-based)
2.1 80	6.4 XScal	10.4 Pentium III
2.2 80	7 32-bit proc	10.5 Pentium II and III Xeon
2.3 80	7.1 80386	10.6 Celeron (Pentium III Coppermine-based)
3 Microcon	7.2 80386	10.7 Celeron (Pentium III Tualatin-based)
3.1 Int	7.3 80387	10.8 Pentium M
3.2 Int	7.4 80387	10.9 Celeron M
3.3 Int	7.5 80387	10.10 Intel Core
3.4 Int	8 32-bit proc	10.11 Dual-Core Xeon LV
3.5 MC	8.1 80486	11 32-bit processors: NetBurst microarchitecture
4 The bit-s	8.2 80486	11.1 Pentium 4
4.1 30	8.3 80486	11.2 Xeon
5 The 16-b	8.4 80486	11.3 Mobile Pentium 4-M
5.1 80	8.5 80486	11.4 Pentium 4 EE
5.2 80	9 32-bit proc	11.5 Pentium 4E
5.3 80	9.1 Orig	11.6 Pentium 4F
5.4 80	9.2 Pent	12 64-bit processors: IA-64
5.5 80		12.1 Itanium
		12.2 Itanium 2



1 The 4-bit	6 32-bit proc	10 13 64-bit processors: Intel 64 – NetBurst microarchitecture
1.1 Int	6.1 iAPX	13.1 Pentium 4F
1.2 Int	6.2 i960	13.2 Pentium D
2 The 8-bit	6.3 i860	13.3 Pentium Extreme Edition
2.1 80	6.4 XScal	13.4 Xeon
2.2 80	7 32-bit proc	14 64-bit processors: Intel 64 – Core microarchitecture
2.3 80	7.1 8038	14.1 Xeon
3 Microcon	7.2 8038	14.2 Intel Core 2
3.1 Int	7.3 8038	14.3 Pentium Dual-Core
3.2 Int	7.4 8038	14.4 Celeron
3.3 Int	7.5 8038	<u>14.5 Celeron M</u>
3.4 Int	8 32-bit proc	15 64-bit processors: Intel 64 – Nehalem microarchitecture
3.5 MC	8.1 8048	15.1 Intel Pentium
4 The bit-s	8.2 8048	15.2 Core i3
4.1 30	8.3 8048	15.3 Core i5
5 The 16-b	8.4 8048	15.4 Core i7
5.1 80	8.5 8048	15.5 Xeon
5.2 80	9 32-bit proc	16 64-bit processors: Intel 64 – Sandy Bridge / Ivy Bridge microarchitecture
5.3 80	9.1 Orig	16.1 Celeron
5.4 80	9.2 Pent	16.2 Pentium
5.5 80		16.3 Core i3
		16.4 Core i5
		16.5 Core i7
		17 64-bit processors: Intel 64 – Haswell microarchitecture

32-bit processors: 386, 486, Pentium, Pentium Pro, Pentium II, Pentium III, Pentium 4, Celeron, Xeon, Itanium, Pentium M, Celeron M, Core 2 Duo, Core Solo, Core i3, Core i5, Core i7, Core i9, Xeon Phi, Xeon E3, Xeon E5, Xeon E7, Xeon W, Xeon D, Xeon S, Xeon F, Xeon G, Xeon H, Xeon I, Xeon J, Xeon K, Xeon L, Xeon M, Xeon N, Xeon O, Xeon P, Xeon Q, Xeon R, Xeon S, Xeon T, Xeon U, Xeon V, Xeon W, Xeon X, Xeon Y, Xeon Z, Xeon AA, Xeon AB, Xeon AC, Xeon AD, Xeon AE, Xeon AF, Xeon AG, Xeon AH, Xeon AI, Xeon AJ, Xeon AK, Xeon AL, Xeon AM, Xeon AN, Xeon AO, Xeon AP, Xeon AQ, Xeon AR, Xeon AS, Xeon AT, Xeon AU, Xeon AV, Xeon AW, Xeon AX, Xeon AY, Xeon AZ, Xeon BA, Xeon BB, Xeon BC, Xeon BD, Xeon BE, Xeon BF, Xeon BG, Xeon BH, Xeon BI, Xeon BJ, Xeon BK, Xeon BL, Xeon BM, Xeon BN, Xeon BO, Xeon BP, Xeon BQ, Xeon BR, Xeon BS, Xeon BT, Xeon BU, Xeon BV, Xeon BW, Xeon BX, Xeon BY, Xeon BZ, Xeon CA, Xeon CB, Xeon CC, Xeon CD, Xeon CE, Xeon CF, Xeon CG, Xeon CH, Xeon CI, Xeon CJ, Xeon CK, Xeon CL, Xeon CM, Xeon CN, Xeon CO, Xeon CP, Xeon CQ, Xeon CR, Xeon CS, Xeon CT, Xeon CU, Xeon CV, Xeon CW, Xeon CX, Xeon CY, Xeon CZ, Xeon DA, Xeon DB, Xeon DC, Xeon DD, Xeon DE, Xeon DF, Xeon DG, Xeon DH, Xeon DI, Xeon DJ, Xeon DK, Xeon DL, Xeon DM, Xeon DN, Xeon DO, Xeon DP, Xeon DQ, Xeon DR, Xeon DS, Xeon DT, Xeon DU, Xeon DV, Xeon DW, Xeon DX, Xeon DY, Xeon DZ, Xeon EA, Xeon EB, Xeon EC, Xeon ED, Xeon EE, Xeon EF, Xeon EG, Xeon EH, Xeon EI, Xeon EJ, Xeon EK, Xeon EL, Xeon EM, Xeon EN, Xeon EO, Xeon EP, Xeon EQ, Xeon ER, Xeon ES, Xeon ET, Xeon EU, Xeon EV, Xeon EW, Xeon EX, Xeon EY, Xeon EZ, Xeon FA, Xeon FB, Xeon FC, Xeon FD, Xeon FE, Xeon FF, Xeon FG, Xeon FH, Xeon FI, Xeon FJ, Xeon FK, Xeon FL, Xeon FM, Xeon FN, Xeon FO, Xeon FP, Xeon FQ, Xeon FR, Xeon FS, Xeon FT, Xeon FU, Xeon FV, Xeon FW, Xeon FX, Xeon FY, Xeon FZ, Xeon GA, Xeon GB, Xeon GC, Xeon GD, Xeon GE, Xeon GF, Xeon GG, Xeon GH, Xeon GI, Xeon GJ, Xeon GK, Xeon GL, Xeon GM, Xeon GN, Xeon GO, Xeon GP, Xeon GQ, Xeon GR, Xeon GS, Xeon GT, Xeon GU, Xeon GV, Xeon GW, Xeon GX, Xeon GY, Xeon GZ, Xeon HA, Xeon HB, Xeon HC, Xeon HD, Xeon HE, Xeon HF, Xeon HG, Xeon HH, Xeon HI, Xeon HJ, Xeon HK, Xeon HL, Xeon HM, Xeon HN, Xeon HO, Xeon HP, Xeon HQ, Xeon HR, Xeon HS, Xeon HT, Xeon HU, Xeon HV, Xeon HW, Xeon HX, Xeon HY, Xeon HZ, Xeon IA, Xeon IB, Xeon IC, Xeon ID, Xeon IE, Xeon IF, Xeon IG, Xeon IH, Xeon II, Xeon IJ, Xeon IK, Xeon IL, Xeon IM, Xeon IN, Xeon IO, Xeon IP, Xeon IQ, Xeon IR, Xeon IS, Xeon IT, Xeon IU, Xeon IV, Xeon IW, Xeon IX, Xeon IY, Xeon IZ, Xeon JA, Xeon JB, Xeon JC, Xeon JD, Xeon JE, Xeon JF, Xeon JG, Xeon JH, Xeon JI, Xeon JJ, Xeon JK, Xeon JL, Xeon JM, Xeon JN, Xeon JO, Xeon JP, Xeon JQ, Xeon JR, Xeon JS, Xeon JT, Xeon JU, Xeon JV, Xeon JW, Xeon JX, Xeon JY, Xeon JZ, Xeon KA, Xeon KB, Xeon KC, Xeon KD, Xeon KE, Xeon KF, Xeon KG, Xeon KH, Xeon KI, Xeon KJ, Xeon KK, Xeon KL, Xeon KM, Xeon KN, Xeon KO, Xeon KP, Xeon KQ, Xeon KR, Xeon KS, Xeon KT, Xeon KU, Xeon KV, Xeon KW, Xeon KX, Xeon KY, Xeon KZ, Xeon LA, Xeon LB, Xeon LC, Xeon LD, Xeon LE, Xeon LF, Xeon LG, Xeon LH, Xeon LI, Xeon LJ, Xeon LK, Xeon LL, Xeon LM, Xeon LN, Xeon LO, Xeon LP, Xeon LQ, Xeon LR, Xeon LS, Xeon LT, Xeon LU, Xeon LV, Xeon LW, Xeon LX, Xeon LY, Xeon LZ, Xeon MA, Xeon MB, Xeon MC, Xeon MD, Xeon ME, Xeon MF, Xeon MG, Xeon MH, Xeon MI, Xeon MJ, Xeon MK, Xeon ML, Xeon MM, Xeon MN, Xeon MO, Xeon MP, Xeon MQ, Xeon MR, Xeon MS, Xeon MT, Xeon MU, Xeon MV, Xeon MW, Xeon MX, Xeon MY, Xeon MZ, Xeon NA, Xeon NB, Xeon NC, Xeon ND, Xeon NE, Xeon NF, Xeon NG, Xeon NH, Xeon NI, Xeon NJ, Xeon NK, Xeon NL, Xeon NM, Xeon NN, Xeon NO, Xeon NP, Xeon NQ, Xeon NR, Xeon NS, Xeon NT, Xeon NU, Xeon NV, Xeon NW, Xeon NX, Xeon NY, Xeon NZ, Xeon OA, Xeon OB, Xeon OC, Xeon OD, Xeon OE, Xeon OF, Xeon OG, Xeon OH, Xeon OI, Xeon OJ, Xeon OK, Xeon OL, Xeon OM, Xeon ON, Xeon OO, Xeon OP, Xeon OQ, Xeon OR, Xeon OS, Xeon OT, Xeon OU, Xeon OV, Xeon OW, Xeon OX, Xeon OY, Xeon OZ, Xeon PA, Xeon PB, Xeon PC, Xeon PD, Xeon PE, Xeon PF, Xeon PG, Xeon PH, Xeon PI, Xeon PJ, Xeon PK, Xeon PL, Xeon PM, Xeon PN, Xeon PO, Xeon PP, Xeon PQ, Xeon PR, Xeon PS, Xeon PT, Xeon PU, Xeon PV, Xeon PW, Xeon PX, Xeon PY, Xeon PZ, Xeon QA, Xeon QB, Xeon QC, Xeon QD, Xeon QE, Xeon QF, Xeon QG, Xeon QH, Xeon QI, Xeon QJ, Xeon QK, Xeon QL, Xeon QM, Xeon QN, Xeon QO, Xeon QP, Xeon QQ, Xeon QR, Xeon QS, Xeon QT, Xeon QU, Xeon QV, Xeon QW, Xeon QX, Xeon QY, Xeon QZ, Xeon RA, Xeon RB, Xeon RC, Xeon RD, Xeon RE, Xeon RF, Xeon RG, Xeon RH, Xeon RI, Xeon RJ, Xeon RK, Xeon RL, Xeon RM, Xeon RN, Xeon RO, Xeon RP, Xeon RQ, Xeon RR, Xeon RS, Xeon RT, Xeon RU, Xeon RV, Xeon RW, Xeon RX, Xeon RY, Xeon RZ, Xeon SA, Xeon SB, Xeon SC, Xeon SD, Xeon SE, Xeon SF, Xeon SG, Xeon SH, Xeon SI, Xeon SJ, Xeon SK, Xeon SL, Xeon SM, Xeon SN, Xeon SO, Xeon SP, Xeon SQ, Xeon SR, Xeon SS, Xeon ST, Xeon SU, Xeon SV, Xeon SW, Xeon SX, Xeon SY, Xeon SZ, Xeon TA, Xeon TB, Xeon TC, Xeon TD, Xeon TE, Xeon TF, Xeon TG, Xeon TH, Xeon TI, Xeon TJ, Xeon TK, Xeon TL, Xeon TM, Xeon TN, Xeon TO, Xeon TP, Xeon TQ, Xeon TR, Xeon TS, Xeon TT, Xeon TU, Xeon TV, Xeon TW, Xeon TX, Xeon TY, Xeon TZ, Xeon UA, Xeon UB, Xeon UC, Xeon UD, Xeon UE, Xeon UF, Xeon UG, Xeon UH, Xeon UI, Xeon UJ, Xeon UK, Xeon UL, Xeon UM, Xeon UN, Xeon UO, Xeon UP, Xeon UQ, Xeon UR, Xeon US, Xeon UT, Xeon UY, Xeon UZ, Xeon VA, Xeon VB, Xeon VC, Xeon VD, Xeon VE, Xeon VF, Xeon VG, Xeon VH, Xeon VI, Xeon VJ, Xeon VK, Xeon VL, Xeon VM, Xeon VN, Xeon VO, Xeon VP, Xeon VQ, Xeon VR, Xeon VS, Xeon VT, Xeon VU, Xeon VV, Xeon VW, Xeon VX, Xeon VY, Xeon VZ, Xeon WA, Xeon WB, Xeon WC, Xeon WD, Xeon WE, Xeon WF, Xeon WG, Xeon WH, Xeon WI, Xeon WJ, Xeon WK, Xeon WL, Xeon WM, Xeon WN, Xeon WO, Xeon WP, Xeon WQ, Xeon WR, Xeon WS, Xeon WT, Xeon WU, Xeon WV, Xeon WW, Xeon WX, Xeon WY, Xeon WZ, Xeon XA, Xeon XB, Xeon XC, Xeon XD, Xeon XE, Xeon XF, Xeon XG, Xeon XH, Xeon XI, Xeon XJ, Xeon XK, Xeon XL, Xeon XM, Xeon XN, Xeon XO, Xeon XP, Xeon XQ, Xeon XR, Xeon XS, Xeon XT, Xeon XU, Xeon XV, Xeon XW, Xeon XX, Xeon XY, Xeon XZ, Xeon YA, Xeon YB, Xeon YC, Xeon YD, Xeon YE, Xeon YF, Xeon YG, Xeon YH, Xeon YI, Xeon YJ, Xeon YK, Xeon YL, Xeon YM, Xeon YN, Xeon YO, Xeon YP, Xeon YQ, Xeon YR, Xeon YS, Xeon YT, Xeon YU, Xeon YV, Xeon YW, Xeon YX, Xeon YY, Xeon YZ, Xeon ZA, Xeon ZB, Xeon ZC, Xeon ZD, Xeon ZE, Xeon ZF, Xeon ZG, Xeon ZH, Xeon ZI, Xeon ZJ, Xeon ZK, Xeon ZL, Xeon ZM, Xeon ZN, Xeon ZO, Xeon ZP, Xeon ZQ, Xeon ZR, Xeon ZS, Xeon ZT, Xeon ZU, Xeon ZV, Xeon ZW, Xeon ZX, Xeon ZY, Xeon ZZ

Product code	Marketing name(s)	Codename(s)
80500	Pentium	P5 (A-step)
80501	Pentium	P5
80502	Pentium	P54C, P54CS
80503	Pentium with MMX Technology	P55C, Tillamook
80521	Pentium Pro	P6
80522	Pentium II	Klamath
80523	Pentium II, Celeron, Pentium II Xeon	Deschutes, Covington, Drake
80524	Pentium II, Celeron	Dixon, Mendocino
80525	Pentium III, Pentium III Xeon	Katmai, Tanner
80526	Pentium III, Celeron, Pentium III Xeon	Coppermine, Cascades
80528	Pentium 4, Xeon	Willamette (Socket 423), Foster
80529	<i>canceled</i>	Timna
80530	Pentium III, Celeron	Tualatin
80531	Pentium 4, Celeron	Willamette (Socket 478)
80532	Pentium 4, Celeron, Xeon	Northwood, Prestonia, Gallatin
80533	Pentium III	Coppermine (cD0-step)
80534	Pentium 4 SFF	Northwood (small form factor)
80535	Pentium M, Celeron M 310–340	Banias
80536	Pentium M, Celeron M 350–390	Dothan
80537	Core 2 Duo T5xxx, T7xxx, Celeron M 5xx	Merom
80538	Core Solo, Celeron M 4xx	Yonah
80539	Core Duo, Pentium Dual-core T-series	Yonah
80541	Itanium	Merced

5.5 80...

Product	Product code	Marketing name(s)	Codename(s)	name(s)
80500	80601	Core i7, Xeon 35xx	Bloomfield	
80501	80602	Xeon 55xx	Gainestown	
80502	80603	Itanium 93xx	Tukwila	
80503	80604	Xeon 65xx, Xeon 75xx	Beckton	
80521	80605	Core i5-7xx, Core i7-8xx, Xeon 34xx	Lynnfield	
80522	80606	<i>canceled</i>	Havendale	
80523	80607	Core i7-7xx QM, Core i7-8xx QM, Core i7-9xx XM	Clarksfield	
80524	80608	<i>canceled</i>	Auburndale	
80525	80609	Atom Z6xx	Lincroft	
80526	80610	Atom N400, D400, D500	Pineview	
80528	80611	<i>canceled</i>	Larrabee	
80529	80612	Xeon C35xx, Xeon C55xx	Jasper Forest	
80530	80613	Core i7-9xxX, Xeon 36xx	Gulftown	
80531	80614	Xeon 56xx	Westmere-EP	
80532	80615	Xeon E7-28xx, Xeon E7-48xx, Xeon E7-88xx	Westmere-EX	
80533	80616	Pentium G6xxx, Core i3-5xx, Core i5-6xx	Clarkdale	
80534	80617	Core i5-5xx, Core i7-6xxM/UM/LM	Arrandale	
80535	80618	Atom E6x0	Tunnel Creek	
80536	80619	Core i7-3xxx	Sandy Bridge-EP	
80537	80620	Xeon E5-24xx	Sandy Bridge-EP-8, Sandy Bridge-EP-4	
80538	80621	Xeon E5-16xx, Xeon E5-26xx, Xeon E5-46xx	Sandy Bridge-EP-8, Sandy Bridge-EP-4	
80539	80622		Sandy Bridge-EP-8	
80541				

5.5 80500

Intel micro-processor history

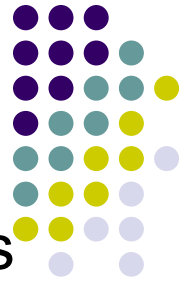
https://en.wikipedia.org/wiki/List_of_Intel_microprocessors



- 2 All processors
 - 2.1 The 4-bit processors
 - 2.1.1 Intel 4004
 - 2.2 The 8-bit processors
 - 2.2.1 8008
 - 2.2.2 8080
 - 2.2.3 8085
 - 2.3 Microcontrollers
 - 2.3.1 Intel 8048
 - 2.3.2 Intel 8051
 - 2.3.3 Intel 80151
 - 2.3.4 Intel 80251
 - 2.3.5 MCS-96 family
 - 2.4 The bit-slice processor
 - 2.4.1 3000 Family
 - 2.5 The 16-bit processors: MCS-86 family
 - 2.5.1 8086
 - 2.5.2 8088
 - 2.5.3 80186
 - 2.5.4 80188
 - 2.5.5 80286
 - 2.6 32-bit processors: the non-x86 microprocessors
 - 2.6.1 iAPX 432
 - 2.6.2 i960 a.k.a. 80960
 - 2.6.3 i860 a.k.a. 80860
 - 2.6.4 XScale
 - 2.7 32-bit processors: the 80386 range
 - 2.7.1 80386DX
 - 2.7.2 80386SX
 - 2.7.3 80376
 - 2.7.4 80386SL
 - 2.7.5 80386EX
 - 2.8 32-bit processors: the 80486 range
 - 2.8.1 80486DX
 - 2.8.2 80486SX
 - 2.8.3 80486DX2
 - 2.8.4 80486SL
 - 2.8.5 80486DX4
 - 2.9 32-bit processors: P5 microarchitecture
 - 2.9.1 Original Pentium
 - 2.9.2 Pentium with MMX Technology
 - 2.10 32-bit processors: P6/Pentium M microarchitecture
 - 2.10.1 Pentium Pro
 - 2.10.2 Pentium II
 - 2.10.3 Celeron (Pentium II-based)
 - 2.10.4 Pentium III
 - 2.10.5 Pentium II Xeon and Pentium III Xeon
 - 2.10.6 Celeron (Pentium III Coppermine-based)
 - 2.10.7 Pentium III Tualatin-based
 - 2.10.8 Celeron (Pentium III Tualatin-based)
 - 2.10.9 Pentium M
 - 2.10.10 Celeron M
 - 2.10.11 Intel Core
 - 2.10.12 Dual-Core Xeon LV
 - 2.11 32-bit processors: NetBurst microarchitecture
 - 2.11.1 Pentium 4
 - 2.11.2 Xeon (32-bit NetBurst)
 - 2.11.3 Mobile Pentium 4-M
 - 2.11.4 Pentium 4 EE
 - 2.11.5 Pentium 4E

Intel micro-processor history

https://en.wikipedia.org/wiki/List_of_Intel_microprocessors



2.12 64-bit processors: IA-64

- 2.12.1 Itanium
- 2.12.2 Itanium 2

2.13 64-bit processors: Intel 64 – NetBurst microarchitecture

- 2.13.1 Pentium 4F
- 2.13.2 Pentium D
- 2.13.3 Pentium Extreme Edition
- 2.13.4 Xeon (64-bit NetBurst)

2.14 64-bit processors: Intel 64 – Core microarchitecture

- 2.14.1 Xeon (64-bit Core microarchitecture)
- 2.14.2 Intel Core 2
- 2.14.3 Intel Pentium Dual-Core
- 2.14.4 Celeron (64-bit Core microarchitecture)
- 2.14.5 Celeron M (64-bit Core microarchitecture)

2.15 64-bit processors: Intel 64 – Nehalem microarchitecture

- 2.15.1 Intel Pentium (Nehalem)
- 2.15.2 Core i3 (1st Generation)
- 2.15.3 Core i5 (1st Generation)
- 2.15.4 Core i7 (1st Generation)
- 2.15.5 Xeon (Nehalem Microarchitecture)

2.16 64-bit processors: Intel 64 – Sandy Bridge / Ivy Bridge microarchitecture

- 2.16.1 Celeron (Sandy Bridge/Ivy Bridge Microarchitecture)
- 2.16.2 Pentium (Sandy Bridge/Ivy Bridge Microarchitecture)
- 2.16.3 Core i3 (2nd and 3rd Generation)
- 2.16.4 Core i5 (2nd and 3rd Generation)
- 2.16.5 Core i7 (2nd and 3rd Generation)

2.17 64-bit processors: Intel 64 – Haswell microarchitecture

- 2.17.1 Core i3 (4th Generation)

2.18 64-bit processors: Intel 64 – Broadwell microarchitecture

- 2.18.1 Core i3 (5th Generation)
- 2.18.2 Core i5 (5th Generation)
- 2.18.3 Core i7 (5th Generation, Including Core-X Series)
- 2.18.4 Other Broadwell CPUs

2.19 64-bit processors: Intel 64 – Skylake microarchitecture

- 2.19.1 Core i3 (6th Generation)
- 2.19.2 Core i5 (6th Generation)
- 2.19.3 Core i7 (6th Generation)
- 2.19.4 Other Skylake Processors

2.20 64-bit processors: Intel 64 – Kaby Lake microarchitecture

2.21 64-bit processors: Intel 64 – Coffee Lake microarchitecture

2.22 64-bit processors: Intel 64 – Cannon Lake microarchitecture

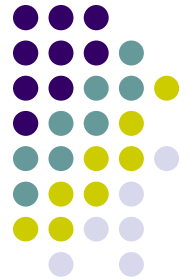
2.23 64-bit processors: Intel 64 – Ice Lake microarchitecture

2.24 Intel Tera-Scale

2.25 Intel 805xx product codes

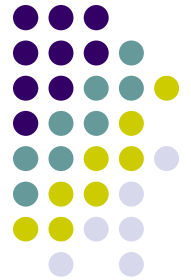
2.26 Intel 806xx product codes

Intel micro-processor series

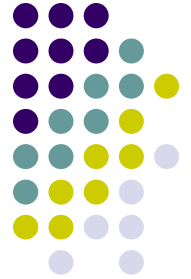


Model	Price (USD)	Cores/Threads	Base frequency (GHz)	Max turbo frequency (GHz)	L3 cache (MB)	Release
i7-8086K	\$425	6/12	4.0	5.0	12	Q2 2018
i7-8700K	\$359	6/12	3.7	4.7	12	Q4 2017
i7-8700	\$303	6/12	3.2	4.6	12	Q4 2017
i5-8600K	\$257	6/6	3.6	4.3	9	Q4 2017
i5-8500	\$202	6/6	3.0	4.1	9	Q2 2018
i5-8400	\$182	6/6	2.8	4.0	9	Q4 2017
i3-8350K	\$168	4/4	4.0	N/A	8	Q4 2017
i3-8100	\$117	4/4	3.6	N/A	6	Q4 2017

Intel micro-processor series



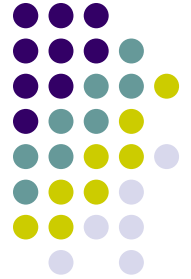
Model	Price (USD)	Cores/Threads	Base frequency (GHz)	Max turbo frequency (GHz)	Release
i9-7980XE	\$1999	18/36	2.6	4.2	Q3 2017 ⁽¹⁾
i9-7960X	\$1699	16/32	2.8	4.2	Q3 2017 ⁽¹⁾
i9-7940X	\$1399	14/28	3.1	4.3	Q3 2017 ⁽¹⁾
i9-7920X	\$1189	12/24	2.9	4.3	Q3 2017
i9-7900X	\$999	10/20	3.3	4.3	Q2 2017
i7-7820X	\$599	8/16	3.6	4.3	Q2 2017
i7-7800X	\$389	6/12	3.5	4.0	Q2 2017
i7-7740X	\$350	4/8	4.3	4.5	Q1 2017
i7-7700K	\$350	4/8	4.2	4.5	Q1 2017
i7-7700	\$312	4/8	3.6	4.2	Q1 2017
i7-7700T	\$312	4/8	2.9	3.8	Q1 2017
i5-7640X	\$242	4/4	4.0	4.2	Q1 2017
i5-7600K	\$243	4/4	3.8	4.2	Q1 2017



Intel-based Assembly

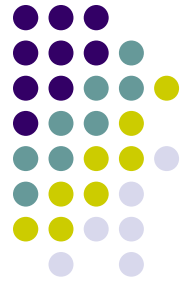
INTEL REGISTERS

Basic Execution Environment



- General-purpose registers
- Index and base registers
- Specialized register uses
- Status flags
- Floating-point, MMX, XMM registers

Some Specialized Register Uses (1 of 2)



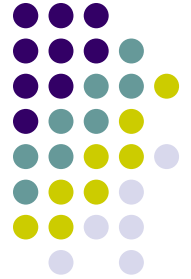
- General-Purpose
 - RAX/EAX – accumulator
 - RCX/ECX – loop counter
 - RSP/ESP – stack pointer
 - RSI/ESI, RDI/EDI – index registers
 - RBP/EBP – extended frame pointer (stack)
- RIP/EIP/IP – instruction pointer
- RFLAGS/EFLAGS
 - status and control flags
 - each flag is a single binary bit

Status Flags



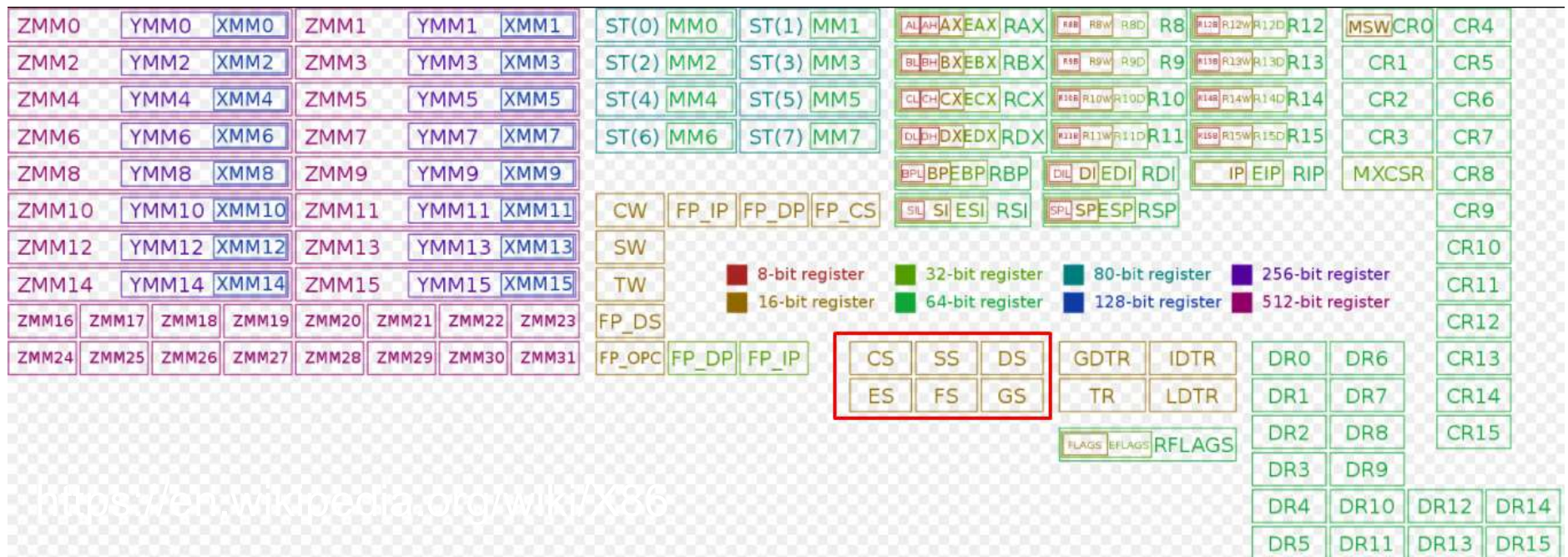
- Carry (CF)
 - unsigned arithmetic out of range
- Overflow (OF)
 - signed arithmetic out of range
- Sign (SF)
 - result is negative
- Zero (ZF)
 - result is zero
- Auxiliary Carry
 - carry from bit 3 to bit 4
- Parity (PF)
 - sum of 1 bits is an even number

X86_64

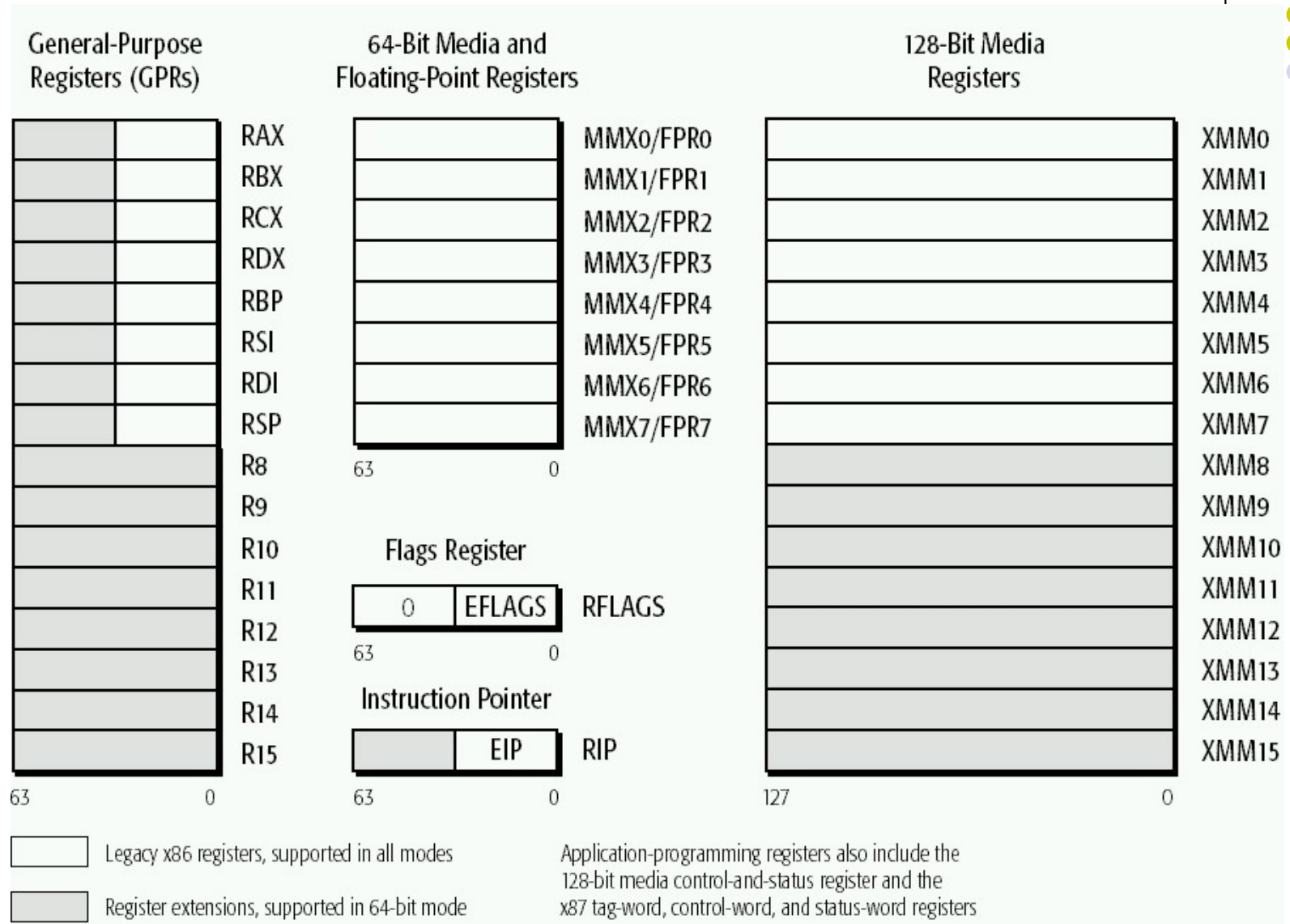


- AMD architecture
 - <http://developer.amd.com/documentation/guides/Pages/default.aspx>
 - <https://software.intel.com/en-us/articles/intel-sdm>
- Expand the registers into 64bits, rax, rbx, rcx, rdx, ...

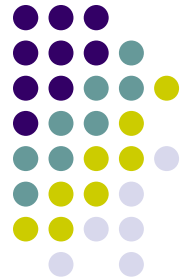
X86-64 Intel registers



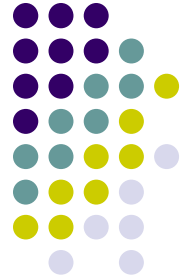
X86_64 registers



X86_64 registers (cont'd)



RAX	EAX	AX	AH	AL
RBX	EBX	BX	BH	BL
RCX	ECX	CX	CH	CL
RDX	EDX	DX	DH	DL
RDI	EDI	DI		DIL
RSI	ESI	SI		SIL
RBP	EBP	BP		BPL
RSP	ESP	SP		SPL
R8	R8D	R8W		R8B
...				
R15	R15D	R15W		R15B



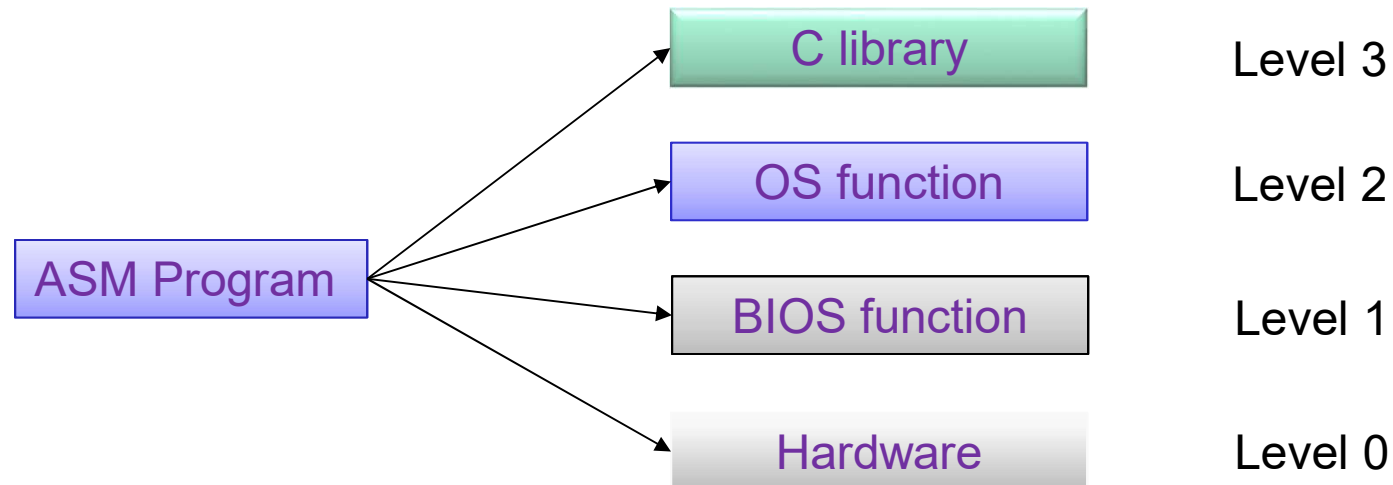
Intel-based Assembly

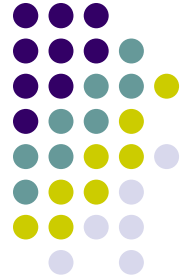
BASIC INSTRUCTIONS



ASM Programming levels

ASM programs can perform input-output at each of the following levels:





Program structure

```
.section .data
```

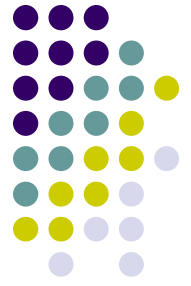
```
output: .asciz "The processor Vendor ID is '%s'\n"
```

```
.section .text
```

```
.globl _start
```

```
_start:
```

```
    program_body
```



Data Definition Statement

- A data definition statement sets aside storage in memory for a variable.
- Syntax:
[name:] directive initializer [,initializer] . . .
- All initializers become binary data in memory
- Data type: .byte, .short (.2byte), .int (.long, .4byte), .quad (.8byte), .float, .double, .asciz, .zero expression

```
value1: .BYTE 'A'           # character constant
value2: .BYTE 0              # smallest unsigned byte
str: .asciz "Hello World"   # string
```

Operand Types



- Three basic types of operands:
 - Immediate – a constant integer
 - Imm8, imm16, imm32, imm64
 - Register – the name of a register
 - register name is converted to a number and encoded within the instruction
 - r8, r16, r32, r64, x (real number processing register)
 - Memory – reference to a location in memory
 - memory address is encoded within the instruction, or a register holds the address of a memory location
 - m8, m16, m32, m64

Instruction Operand Notation

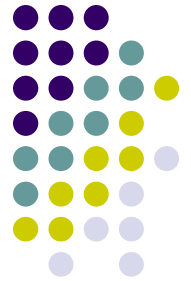


Convention:

- w (word): 16 bits;
- d (double word): 32 bits;
- q (quadword): 64 bits

Operand	Description
r8	8-bit general purpose register: AH, AL, BH, BL, CH, CL, r8b, ...
r16	16 bit general purpose register: AX, BX, CX, DX, SI, DI, r8w, ...
r32	32 bit general purpose register: EAX, EBX, ECX, EDX, r8d, ...
r64	64 bit general purpose register: RAX, RBX, RCX, RDX, r8, ...
imm8/16/32/64	An immediate of 8, 16, 32, 63 bit
m8/16/32/64	A variable of 8, 16, 32, 64 bit
x	xmm register
r/m8/16/32/64	A register or variable of 8, 16, 32, 64
reg	any general purpose register

Assembly standards



- Intel standard

```
mov dst, src
mov eax, 4
add ebx, 1
sub ecx, ebx
```

ADD AL, *imm8*

ADD AX, *imm16*

ADD EAX, *imm32*

ADD RAX, *imm32*

ADD *r/m8*, *imm8*

ADD *r/m8**, *imm8*

ADD *r/m16*, *imm16*

ADD *r/m32*, *imm32*

ADD *r/m64*, *imm32*

- AT&T standard

```
mov src, dst
mov $4, %eax
add $1, %ebx
sub %ebx, %ecx
```

movss

movss

movsd

movsd

M_{32}/X

X

M_{64}/X

X

X

M_{32}

X

M_{64}

Manual



ADD—Add

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
04 <i>ib</i>	ADD AL, <i>imm8</i>	I	Valid	Valid	Add <i>imm8</i> to AL.
05 <i>iw</i>	ADD AX, <i>imm16</i>	I	Valid	Valid	Add <i>imm16</i> to AX.
05 <i>id</i>	ADD EAX, <i>imm32</i>	I	Valid	Valid	Add <i>imm32</i> to EAX.
REX.W + 05 <i>id</i>	ADD RAX, <i>imm32</i>	I	Valid	N.E.	Add <i>imm32</i> sign-extended to 64-bits to RAX.
80 /0 <i>ib</i>	ADD r/m8, <i>imm8</i>	MI	Valid	Valid	Add <i>imm8</i> to r/m8.
REX + 80 /0 <i>ib</i>	ADD r/m8*, <i>imm8</i>	MI	Valid	N.E.	Add sign-extended <i>imm8</i> to r/m64.
81 /0 <i>iw</i>	ADD r/m16, <i>imm16</i>	MI	Valid	Valid	Add <i>imm16</i> to r/m16.
81 /0 <i>id</i>	ADD r/m32, <i>imm32</i>	MI	Valid	Valid	Add <i>imm32</i> to r/m32.
REX.W + 81 /0 <i>id</i>	ADD r/m64, <i>imm32</i>	MI	Valid	N.E.	Add <i>imm32</i> sign-extended to 64-bits to r/m64.

Description

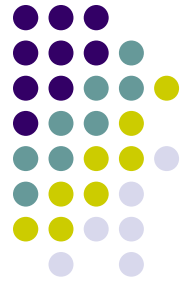
Adds the destination operand (first operand) and the source operand (second operand) and then stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one instruction.) When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

Operation

DEST ← DEST + SRC;

Flags Affected

The OF, SF, ZF, AF, CF, and PF flags are set according to the result.



MOV Instruction (assignment)

- Move from source to destination. Syntax:

MOV source, destination

- Both operands must be the same size
- No more than one memory operand permitted

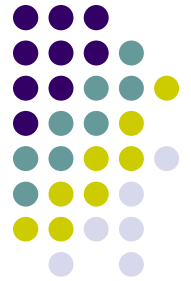
```
.section .data
Output: .asciz "The result is: "
Val: .int 10
.section text
...
mov $4, %eax      #eax=4
mov $1, %ebx      #ebx=1
mov $output, %rcx #rcx=&output
mov $12, %edx     #edx=12
...
mov val, %eax     #eax=val
...
mov %eax, val     #val=eax
```



Direct-Offset Operands

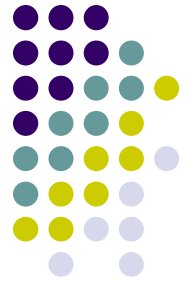
An offset is added to a data label to produce an effective address (EA).

```
arr: .int 34,3,12,4,3,5
arrB:.byte 1, 2, 3, 4
...
xor %edx,%edx           #edx=0
mov arr(,%edx,4),%ebx    #ebx=arr[edx]; ebx = ?
inc %edx
...
mov %ah,arrB(,%ebx,1)    #arrB[ebx]=ah
mov %al,[arrB+1]         # alternative notation
```

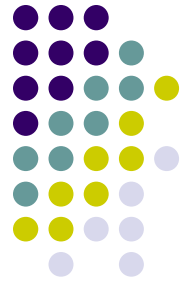
Addition and Subtraction

- INC and DEC Instructions
- ADD and SUB Instructions
- NEG Instruction
- Implementing Arithmetic Expressions
- Flags Affected by Arithmetic
 - Zero
 - Sign
 - Carry
 - Overflow



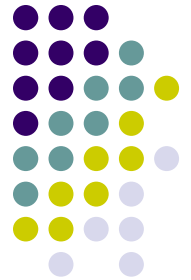
INC and DEC Instructions

- Add 1, subtract 1 from destination operand
 - operand may be register or memory
- INC *destination*
 - Logic: $destination \leftarrow destination + 1$
- DEC *destination*
 - Logic: $destination \leftarrow destination - 1$



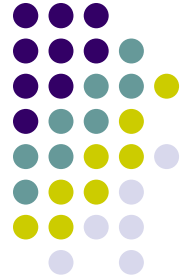
ADD and SUB Instructions

- ADD source, destination
 - Logic: $destination \leftarrow destination + source$
- SUB source, destination
 - Logic: $destination \leftarrow destination - source$
- Same operand rules as for the MOV instruction



ADD and SUB Examples

```
var1: .int 0x10000
var2: .int 0x20000
...
mov var1, %eax      #eax=var1; 00010000h
mov var2, %ebx      #ebx=var2;
add %ebx,%eax,      #eax+=ebx; 00030000h
add $0xFFFF,%ax    #ax+=0xFFFF; eax=0003FFFFh
add $1,%eax         #eax+=1;
sub $1,%ax          #ax-=1;
neg %eax            #eax=-eax;
```



NEG (negate) Instruction

Reverses the sign of an operand. Operand can be a register or memory operand.

- NEG destination
 - Logic: $destination \leftarrow - destination$

```
valB: .BYTE -1
valW .int +32767
...
mov valB,%al  # AL = -1
neg %al       # AL = +1
neg valW      # valW = -32767
```



MUL Instruction

- Unsigned multiplication
- MUL r8/m8 - MUL r16/m16
- MUL r32/m32 - MUL r64/m64

Multiplicant	Multiplier	Product
AL	r8/m8	AX
AX	r16/m16	DX:AX
EAX	r32/m32	EDX:EAX
RAX	r64/m64	RDX:RAX

Homework: study imul instruction for signed numbers



DIV Instruction

- Unsigned multiplication
- DIV r8/m8 - DIV r16/m16 - DIV r32/m32 - DIV r64/m64

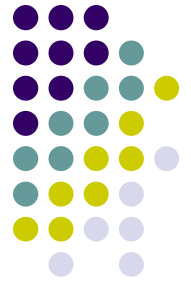
Dividend	Divisor	Quotient	Remainder
AX	r8/m8	AL	AH
DX:AX	r16/m16	AX	DX
EDX:EAX	r32/m32	EAX	EDX
RDX:RAX	r64/m64	RAX	RDX

- Division preparation: zero upper registers

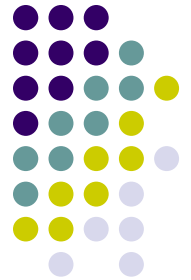
Instruction	Meaning
CBW	AX=SE(AL)
CWD	DX:AX=SE(AX)
CDQ	EDX:EAX=SE(EAX)
CQO	RDX:RAX=SE(RAX)

Homework: study idiv instruction for signed numbers. Use left instructions for preparation

Flags Affected by Arithmetic



- The ALU has a number of status flags that reflect the outcome of arithmetic (and bitwise) operations
 - based on the contents of the destination operand
- Essential flags:
 - Zero flag – set when destination equals zero
 - Sign flag – set when destination is negative
 - Carry flag – set when unsigned value is out of range
 - Overflow flag – set when signed value is out of range
- The MOV instruction never affects the flags.



Zero Flag (ZF)

The Zero flag is set when the result of an operation produces zero in the destination operand.

```
mov $1,%cx    # no change in flags
sub $1,%cx    # CX = 0, ZF = 1
mov $0xFFFF,%ax
inc %ax       # AX = 0, ZF = 1
inc %ax       # AX = 1, ZF = 0
```

Remember...

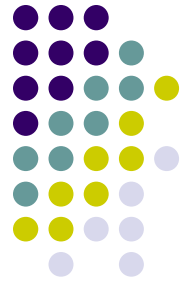
- A flag is **set** when it equals 1.
- A flag is **clear** when it equals 0.

JMP Instruction



- JMP is an unconditional jump to a label that is usually within the same procedure.
- Syntax: **JMP** *target*
- Logic: $RIP \leftarrow target$
- Example:

```
top:
.  
.  
jmp top  #goto top
```

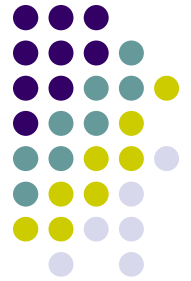


JMP Instruction

- JMP is an unconditional jump to a label that is usually within the same procedure.
- Syntax: **JMP** *target*
- Logic: $RIP \leftarrow target$
- Example:

```
top:
.  
.  
jmp top
```

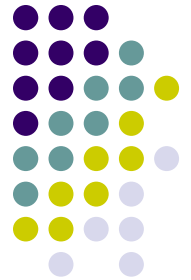
A jump outside the current procedure must be to a special type of label called a **global label** (see Section 5.5.2.3 for details).



TEST Instruction

- Performs a nondestructive AND operation between each pair of matching bits in two operands
- No operands are modified, but the Zero flag is affected.
- Example: jump to a label if either bit 0 or bit 1 in AL is set.

```
test $11,%al  
jnz  ValueFound
```

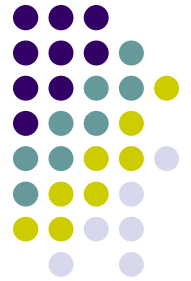


CMP Instruction (1 of 3)

- Compares the destination operand to the source operand
 - Nondestructive subtraction of source from destination (destination operand is not changed)
 - The **flags** will be affected
 - One source or destination can be an **immediate**
- Syntax: **CMP** *source, destination*

```
mov $5,%a1  
cmp %a1,%b1    # Zero flag set
```

- Example: destination == source?



***Jcond* Instruction**

- A conditional jump instruction branches to a label when specific register or flag conditions are met
- Examples:
 - JB, JC jump to a label if the Carry flag is set
 - JE, JZ jump to a label if the Zero flag is set
 - JS jumps to a label if the Sign flag is set
 - JNE, JNZ jump to a label if the Zero flag is clear
 - JRCXZ (JECXZ or JCXZ) jumps to a label if RCX (ECX or CX) equals 0

Jumps Based on Specific Flags



Instruction	Description	C instruction
JZ label	Jump if zero	if(ZF==1) goto label;
JNZ label	Jump if not zero	if(ZF==0) goto label;
JC label	Jump if carry	if(CF==1) goto label;
JNC label	Jump if not carry	if(CF==0) goto label;
JO label	Jump if overflow	if(OF==1) goto label;
JNO label	Jump if not overflow	if(OF==0) goto label;
JS label	Jump if signed	if(SF==1) goto label;
JNS label	Jump if not signed	if(SF==0) goto label;
JP label	Jump if parity (even)	if(PF==1) goto label;
JNP label	Jump if not parity (odd)	if(PF==0) goto label;

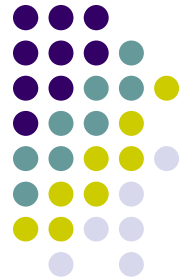


Jumps Based on Equality

cmp left, right

Instruction	Description
JE label	if(right==left) goto label;
JNE label	if(right!=left) goto label;
JCXZ label	if (%CX==0) goto label;
JECXZ	if (%ECX==0) goto label;
JRCXZ	if (%RCX==0) goto label;

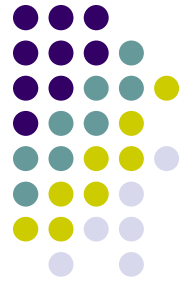
Jumps Based on Unsigned Comparisons



cmp left, right

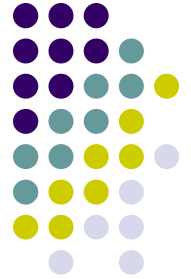
Mnemonic	Description	Flag
JA <i>label</i>	if (<i>right</i> > <i>left</i>) goto <i>label</i> ;	CF=0 && ZF=0
JNBE <i>label</i>	if (<i>right</i> > <i>left</i>) goto <i>label</i> ;	CF=0 && ZF=0
JAЕ <i>label</i>	if (<i>right</i> >= <i>left</i>) goto <i>label</i> ;	CF=0
JNB <i>label</i>	if (<i>right</i> >= <i>left</i>) goto <i>label</i> ;	CF=0
JB <i>label</i>	if (<i>right</i> < <i>left</i>) goto <i>label</i> ;	CF=1
JNAЕ <i>label</i>	if (<i>right</i> < <i>left</i>) goto <i>label</i> ;	CF=1
JBE <i>label</i>	if (<i>right</i> <= <i>left</i>) goto <i>label</i> ;	CF=1 && ZF=1
JNA <i>label</i>	if (<i>right</i> <= <i>left</i>) goto <i>label</i> ;	A=Above; E=Equal;
JE <i>label</i>	if (<i>right</i> == <i>left</i>) goto <i>label</i> ;	N=Not; J=Jump
JNE <i>label</i>	if (<i>right</i> != <i>left</i>) goto <i>label</i> ;	B=Below

Jumps Based on Signed Comparisons



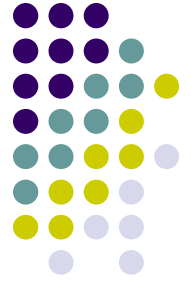
cmp left, right

Mnemonic	Description	Flag
JG <i>label</i>	if (<i>right</i> > <i>left</i>) goto <i>label</i> ;	SF=OF && ZF=0
JNLE <i>label</i>	if (<i>right</i> > <i>left</i>) goto <i>label</i> ;	SF=OF && ZF=0
JGE <i>label</i>	if (<i>right</i> >= <i>left</i>) goto <i>label</i> ;	SF=OF
JNL <i>label</i>	if (<i>right</i> >= <i>left</i>) goto <i>label</i> ;	SF=OF
JL <i>label</i>	if (<i>right</i> < <i>left</i>) goto <i>label</i> ;	SF!=OF
JNGE <i>label</i>	if (<i>right</i> < <i>left</i>) goto <i>label</i> ;	SF!=OF
JLE <i>label</i>	if (<i>right</i> <= <i>left</i>) goto <i>label</i> ;	SF!=OF && ZF=1
JNG <i>label</i>	if (<i>right</i> <= <i>left</i>) goto <i>label</i> ;	SF!=OF && ZF=1
JE <i>label</i>	if (<i>right</i> == <i>left</i>) goto <i>label</i> ;	G=Greater than
JNE <i>label</i>	if (<i>right</i> != <i>left</i>) goto <i>label</i> ;	L=Less than



Intel-based Assembly

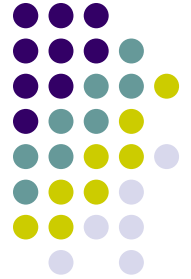
CONTROL STRUCTURE



Conditional Structures

- Block-Structured IF Statements
- Compound Expressions with AND
- Compound Expressions with OR
- WHILE Loops
- Table-Driven Selection

Applications

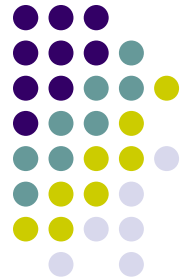


- Task: Jump to a label if **unsigned** EAX is greater than EBX
- Solution: Use CMP, followed by JA

```
cmp %ebx, %eax  
ja  Larger
```

```
if (%eax > %ebx)  
    goto Larger
```

Applications



- Task: Jump to a label if **unsigned** EAX is greater than EBX
- Solution: Use CMP, followed by JA

```
cmp %ebx, %eax  
ja  Larger
```

```
if (%eax > %ebx)  
    goto Larger
```

- Task: Jump to a label if **signed** EAX is greater than EBX
- Solution: Use CMP, followed by JG

```
cmp %ebx, %eax  
jg  Greater
```

```
if (%eax > %ebx)  
    goto Larger
```

Block-Structured IF Statements



Assembly language programmers can easily translate logical statements written in C++/Java into assembly language. For example:

```
if( op1 == op2 )  
    X = 1;  
else  
    X = 2;
```

```
        mov op1,%eax  
        mov op2,%ebx  
if:  
        cmp %ebx,%eax  
        jne else  
then:  mov $1,X  
        jmp endif  
else:  mov $2,X  
endif:
```

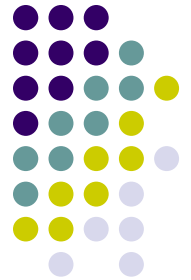


Your turn . . .

Implement the following pseudocode in assembly language. All values are unsigned:

```
if(ebx <= ecx )  
{  
    eax = 5;  
    edx = 6;  
}
```

(There are multiple correct solutions to this problem.)



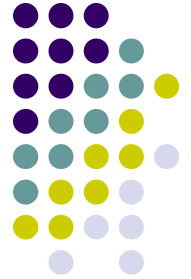
Your turn . . .

Implement the following pseudocode in assembly language. All values are unsigned:

```
if(ebx <= ecx )  
{  
    eax = 5;  
    edx = 6;  
}
```

```
if:    cmp %ecx,%ebx  
        ja  endif  
then:  mov $5, %eax  
        mov $6,%edx  
endif:
```

(There are multiple correct solutions to this problem.)



Your turn . . .

Implement the following pseudocode in assembly language. All values are unsigned:

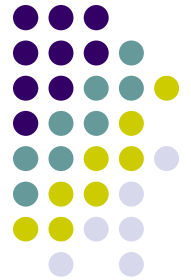
```
if(ebx <= ecx )  
{  
    eax = 5;  
    edx = 6;  
}
```

```
if:cmp %ecx,%ebx  
    ja  endif  
    mov $5,%eax  
    mov $6,%edx  
endif:
```

```
if: cmp %ecx,%ebx  
    jbe then  
    jmp endif  
then: mov $5,%eax  
      mov $6,%edx  
endif:
```

(There are multiple correct solutions to this problem.)

Compound Expression with AND (2 of 3)

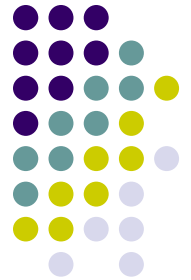


```
if ((a1 > b1) && (b1 > c1))  
    X = 1;
```

This is one possible implementation . . .

```
if: cmp %b1,%a1                # first expression...  
    ja  L1  
    jmp endif  
L1:                               # second expression...  
    cmp %c1,%b1  
    ja  L2  
    jmp endif  
L2:                               # both are true  
    mov $1,X                   # set X to 1  
endif:
```

Compound Expression with AND (3 of 3)



```
if ((a1 > b1) && (b1 > c1))  
    x = 1;
```

But the following implementation uses 29% less code by reversing the first relational operator. We allow the program to "fall through" to the second expression:

```
if:  cmp %b1,%a1          # first expression...  
     jbe endif           # quit if false  
     cmp %c1,%b1         # second expression...  
     jbe endif           # quit if false  
then: mov $1,x           # both are true  
endif:
```



Your turn . . .

Implement the following pseudocode in assembly language. All values are unsigned:

```
if( ebx <= ecx
  && ecx > edx )
{
    eax = 5;
    edx = 6;
}
```

(There are multiple correct solutions to this problem.)



Your turn . . .

Implement the following pseudocode in assembly language. All values are unsigned:

```
if( ebx <= ecx
  && ecx > edx )
{
    eax = 5;
    edx = 6;
}
```

```
if: cmp %ebx, %ecx
    ja next
    cmp %ecx, %edx
    jbe next
    mov $5, %eax
    mov $6, %edx
next:
```

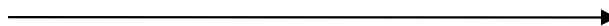
(There are multiple correct solutions to this problem.)

Compound Expression with OR (1 of 2)

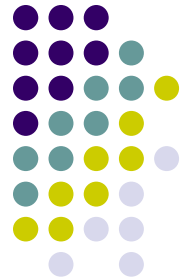


- When implementing the logical OR operator, consider that HLLs use short-circuit evaluation
- In the following example, if the first expression is true, the second expression is skipped:

```
if ((a1 > b1) || (b1 > c1))  
    x = 1;
```



Compound Expression with OR (1 of 2)

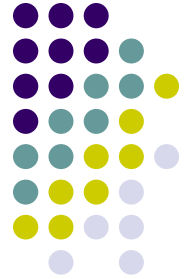


```
if ((a1 > b1) || (b1 > c1))  
    X = 1;
```

We can use "fall-through" logic to keep the code as short as possible:

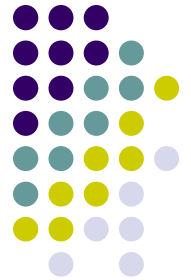
```
if:cmp %b1,%a1          # is AL > BL?  
    ja  then            # yes  
    cmp %c1,%b1         # no: is BL > CL?  
    jbe endif           # no: skip next statement  
then:mov $1, X          # set X to 1  
endif:
```


WHILE Loops



A WHILE loop is really an IF statement followed by the body of the loop, followed by an unconditional jump to the top of the loop. Consider the following example:

```
while( eax < ebx)
    eax = eax + 1;    #do
```



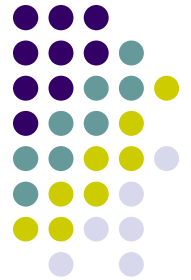
WHILE ... DO Loops

A WHILE loop is really an IF statement followed by the body of the loop, followed by an unconditional jump to the top of the loop. Consider the following example:

```
while( eax < ebx) {  
    eax = eax + 1;  
}
```

This is a possible implementation:

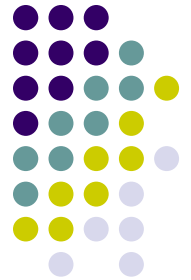
```
while: cmp %ebx, %eax      # check loop condition  
      jae endwhile        # false? exit loop  
do:   inc %eax            # body of loop  
      jmp while           # repeat the loop  
endwhile:
```



Your turn . . .

Implement the following loop, using unsigned 32-bit integers:

```
while( ebx <= val1)
{
    ebx = ebx + 5;
    val1 = val1 - 1;
}
```



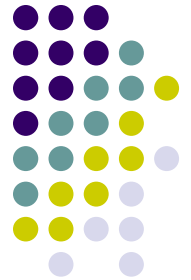
Your turn . . .

Implement the following loop, using unsigned 32-bit integers:

```
while( ebx <= val1)
{
    ebx = ebx + 5;
    val1 = val1 - 1;
}
```

```
while: cmp val1,%ebx      # check loop condition
        ja  endwhile     # false? exit loop
do:     add $5,%ebx        # body of loop
        dec val1
        jmp while         # repeat the loop
endwhile:
```

DO ...WHILE Loops



```
r8=0; rax=0;
do{
    rax++;
    r8+= rax;
}while(rax < rbx);
```

This is a possible implementation:

```
    xor %r8,%r8
    xor %rax,%rax
do:
    inc %rax                # body of loop
    add %rax,%r8
while:
    cmp %rbx,%rax          # check loop condition
    jb do                  # exit loop or repeat
```



LOOP Instruction- for loop

- The LOOP instruction creates a counting loop
- Syntax: `LOOP target`
- Logic:
 - $ECX \leftarrow ECX - 1$
 - if $ECX \neq 0$, jump to *target*
- Implementation:
 - The assembler calculates the distance, in bytes, between the offset of the following instruction and the offset of the target label. It is called the **relative offset**.
 - The relative offset is added to EIP.



LOOP Instruction- for loop

Calculate the total of the first n integers ($n > 0$)

```
for (r9d=0,ecx=n;ecx>0;ecx--) r9d+=ecx;
```

```
init:
    mov $0,%r9d
    mov n,%ecx
for:
    add %ecx,%r9d
    loop for
```

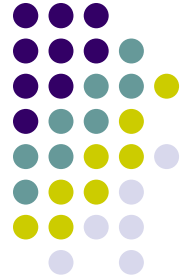


for loop – general case

Calculate the total of the first n integers ($n > 0$)

```
for (r9d=0 , r10d=0 ; r10d<=n ; r10d++)  r9d+=  
    r10d;
```

```
init:  
    mov $0,%r9d  
    xor %r10d,%r10d  
while:  
    cmp n,%r10d  
    ja endwhile  
do:  
    add %r10d,%r9d  
    inc %r10d  
    jmp while  
endwhile:
```

Intel-based Assembly

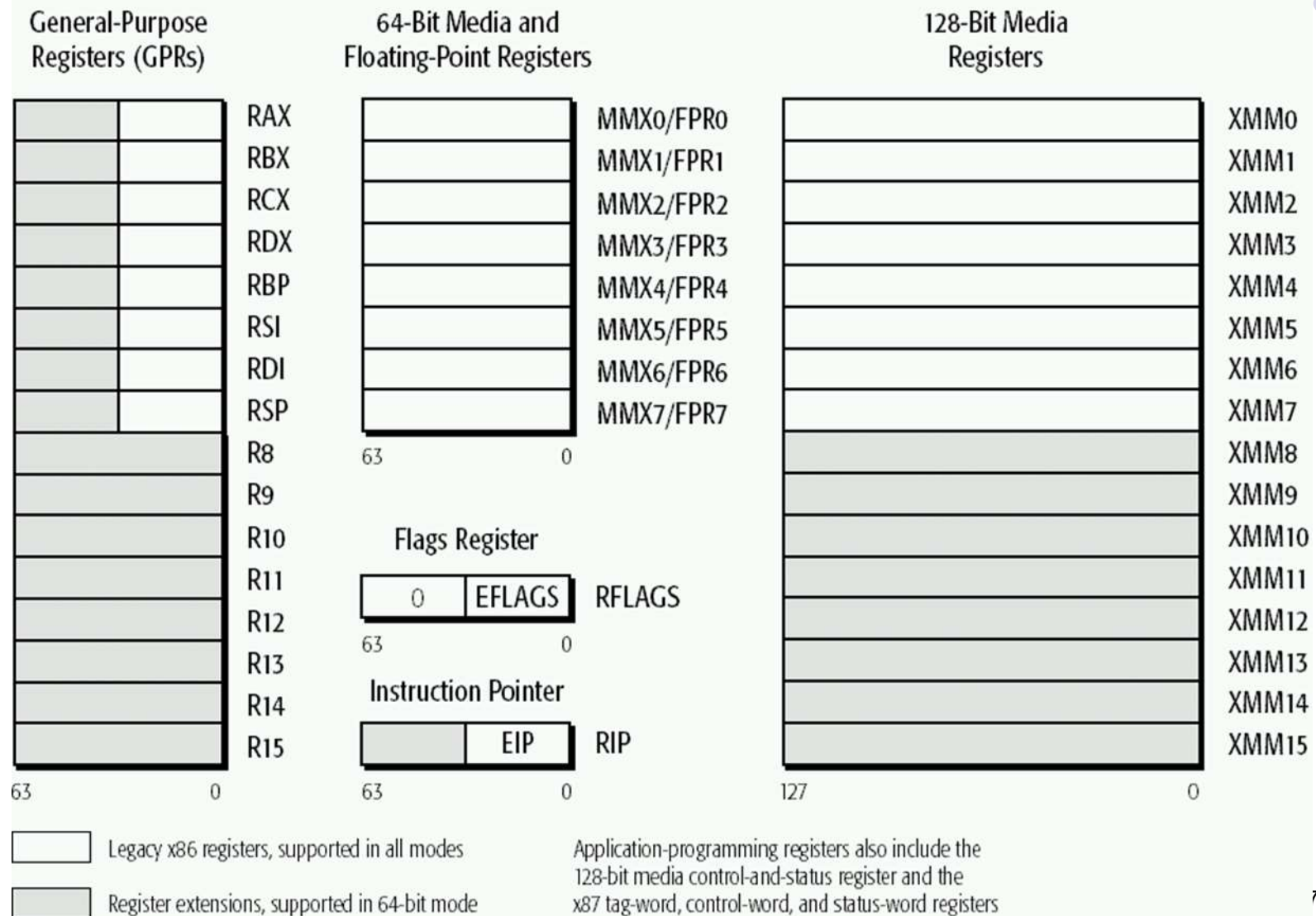
REAL NUMBER MANIPULATION

Streaming SIMD Extension (SSE)

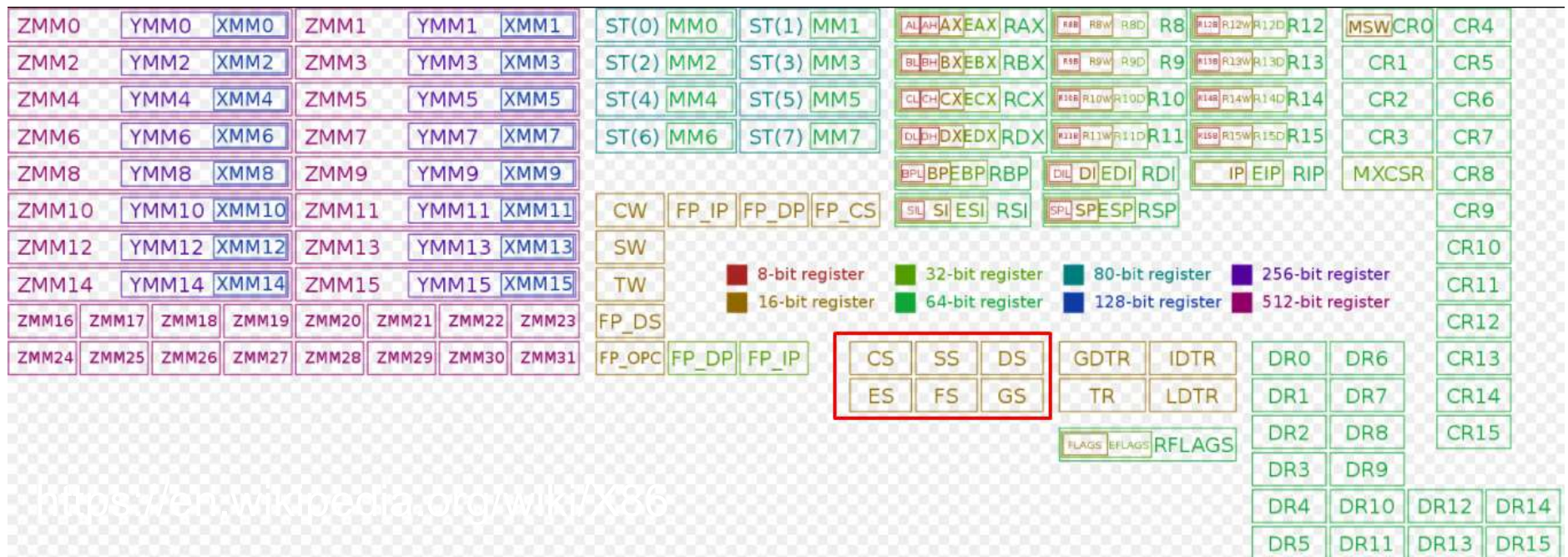


- Use 16× 128-bit registers
- Can be used for multiple FP operands
 - 2 × 64-bit double precision
 - 4 × 32-bit single precision
 - Instructions operate on them simultaneously
 - Single-Instruction Multiple-Data
- SSE4 (version 4) is now **available**

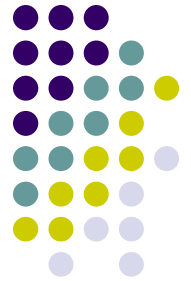
SSE introduction



X86-64 Intel registers



SSE instructions: assignment



Instruction	Source	Destination	Description
movss	M32/X	X	dst=src;
movss	X	M32	dst=src;
movsd	M64/X	X	dst=src;
movsd	X	M64	dst=src;

X: XMM register (e.g., %xmm3)

R32: 32-bit general purpose register (e.g., %eax)

R64: 64-bit general purpose register (e.g., %rax)

M32: 32-bit variable / memory range

M64: 64-bit variable / memory range



SSE instructions (cont'd)

```
len: .double 23.45
result: .double 0.0
arr: .double 3.1,2.3,3.4,4.5,5.6
...
movsd len,%xmm0
movsd %xmm0,result
mov $1, %edx
movsd arr(,%edx,8),%xmm1
```

SSE instructions (cont'd)



float	double	Description
addss src, dst	addsd src, dst	dst+=src;
subss src, dst	subsd src, dst	dst-=src;
mulss src, dst	mulsd src, dst	dst*=src;
divss src, dst	divsd src, dst	dst/=src;
maxss src, dst	maxsd src, dst	dst=max(src,dst);
minss src, dst	minsd src, dst	dst=min(src, dst);
sqrtps src, dst	sqrtsd src, dst	dst=sqrt(src);

xorps S,D	$D \leftarrow D \text{ xor } S$	S, D are xmm registers
ucomiss left,right	like cmp left, right	Compare single precision
ucomisd left,right	like cmp left, right	Compare double precision

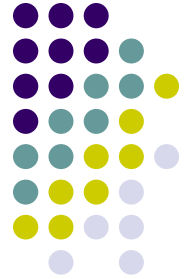
Use JA, JB, JAE, JBE, JE, JNE to make a branch, s is an xmm or a variable



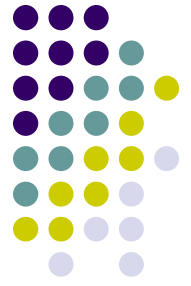
SSE instructions (cont'd)

```
len: .double 23.45
result: .double 0.0
arr: .double 3.1,2.3,3.4,4.5,5.6
...
movsd len,%xmm0
movsd arr(,%edx,8),%xmm1
addsd %xmm1,%xmm0
movsd %xmm0,result
```


SSE instructions (cont'd)

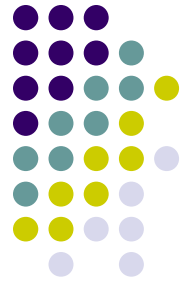


```
...  
    ucomisd %xmm1, %xmm0  
    jb else  
    movsd %xmm1,%xmm0  
else:  
    movsd %xmm0,result  
...
```



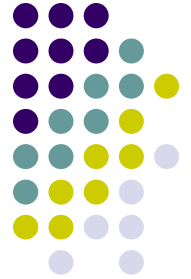
Exercises

- Write a program to add two double numbers and print the result on screen
- Write a program to multiply two double numbers and print the result on screen
- Write a program to print the maximum number of the two double numbers
- Write a program to sum the elements of a double array and print the result on screen



Exercises (cont'd)

- Write a program to solve the equation $ax+b=0$
- Write a program to solve the equation $ax^2+bx+c=0$
- Write a program to print the first of n numbers of a *geometric sequence* (cấp số nhân) with a given value of a and r
- Write a program to print the first of n number in an *arithmetic sequence* (cấp số cộng) with a given value of d and u
- Write a program to find the maximum number of a double array



Intel-based Assembly

NUMERIC TYPE CONVERSION

Numeric types and conversions



- There are a number of numeric types
 - char, unsigned char, int, unsigned int, short, unsigned short, long, unsigned long, long long, unsigned long long, float, double
- There are pointers to the above types
 - how to handle these complexity



Linux 64bit C data model

Integer data type (in bits)

Model	char	short	int	long	pointer (long)
LP64	8	16	32	64	64

SSE: real-2-real, integer-2-real conversion



Instruction	Source	Destination	Description
cvtss2sd	M32/X	X	dst=double(src); //src is float
cvtsd2ss	M64/X	X	dst=float(src); //src is double
cvtsi2ss	M32/R32	X	dst=float(src); //src is int
cvtsi2sd	M32/R32	X	dst=double(src); //src is int
cvtsi2ssq	M64/R64	X	dst=float(src); //src is long
cvtsi2sdq	M64/R64	X	dst=double(src); //src is long

X: XMM register (e.g., %xmm3)

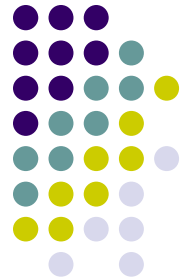
R32: 32-bit general purpose register (e.g., %eax), or int

R64: 64-bit general purpose register (e.g., %rax), or long

M32: 32-bit variable / memory range, of int or float

M64: 64-bit variable / memory range, of long or double

SSE: real-2-integer conversion



Instruction	Source	Destination	Description
cvttss2si	M32/X	R32	dst=int(src); //src is float
cvttss2si	M64/X	R32	dst=int(src); //src is double
cvttss2siq	M32/R32	R64	dst=long(src); //src is float
cvttss2siq	M64/R64	R64	dst=long(src) //src is double

X: XMM register (e.g., %xmm3)

R32: 32-bit general purpose register (e.g., %eax)

R64: 64-bit general purpose register (e.g., %rax)

M32: 32-bit variable / memory range

M64: 64-bit variable / memory range

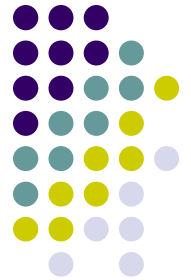
unsigned Integer data conversion



unsigned char uc=12; unsigned short us=71;
unsigned int ui = 23; unsigned long ul=98;

Instruction	Description	Example
movzx r/m8, r16	us=unsigned short(uc);	movzx uc, %ax
movzx r/m8, r32	ui=unsigned int(uc);	movzx uc, %eax
movzx r/m8, r64	ul=unsigned long(uc);	movzx uc, %rax
movzx r/m16, r32	ui=unsigned int(us);	movzx us, %eax
mov r/m16, r64	ul=unsigned long(us);	movzx us, %rax
mov r/m32, r32	ul=unsigned long(ui);	mov ui, %eax #(*)

(*) Upper 32 bits of %rax will be filled by 0
=> %rax = unsigned long(ui);

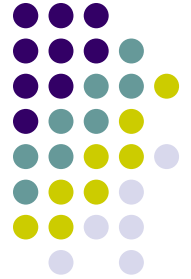


Signed Integer data conversion

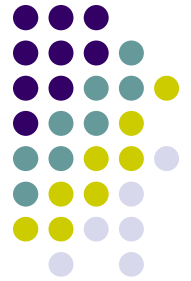
```
char c=-12; short s=71;  
int i = -23; long l=98;
```

Instruction	Description	Example
movsx r/m8, r16	s=short(c);	movsx c, %ax
movsx r/m8, r32	i=int(c);	movsx c, %eax
movsx r/m8, r64	l=long(c);	movsx c, %rax
movsx r/m16, r32	i=int(s);	movsx s, %eax
movsx r/m16, r64	l=long(s);	movsx s, %rax
movsxd r/m32, r64	l=long(i);	movsxd i, %rax

Signed Integer data conversion



Instruction	Description
CBW	AX=SE(AL)
<u>CWDE</u>	EAX=SE(AX)
CDQE	RAX=SE(EAX)

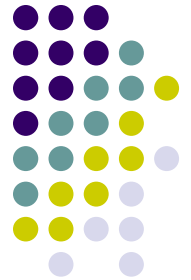


Bigger to smaller Integer conversion

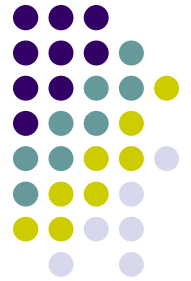
char c=-12; short s=71;
int i = -23; long l=98;
unsigned char uc=12; unsigned short us=71;
unsigned int ui = 23; unsigned long ul=98;

Instruction	Description
mov ul, %rax	ui=%eax; us= %ax; uc=%al;
mov l, %rax	i=%eax; s= %ax; c=%al;
mov ui, %eax	us= %ax; uc=%al;
mov l, %eax	s= %ax; c=%al;
mov us, %ax	uc=%al;
mov s, %ax	c=%al;

Integer conversions



```
i: .int -6
l: .long # l=long(i); is translated as
msg: .asciz "long value is %ld"
...
mov i , %eax
movsxd %eax, %rsi #conversion
mov %rsi, l
mov $msg, %rdi
call printf
```

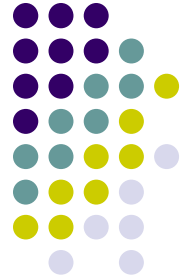


Unsigned Integer conversions

```
ui: .int 0xFFAABBCC #unsigned int ui;  
ul: .long # unsigned long ul;  
msg: .asciz "ulong value is %lu"  
#l=unsigned long(ui); is translated as
```

...

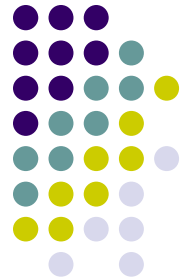
```
mov ui , %eax  
mov %eax, %esi #conversion  
mov %rsi, l  
mov $msg, %rdi  
call printf # 0xFFAABBCC
```



Intel-based Assembly

FUNCTION/PROCEDURE

Procedure/Function



- Define

`convert:`

`mov $10,%ebx`

`xor %ecx, %ecx`

`...`

`ret`

Parameters are passed via registers

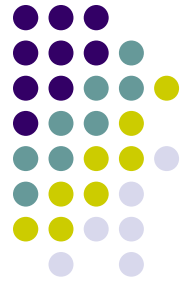
- Call

`call convert`

Steps to call:

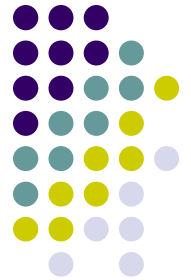
1. Assign parameters to suitable registers
2. call proc/funct
3. Use the returned value

C library function arguments



Real arguments: 1. xmm0, 2. xmm1, ...; return value xmm0

64	32	16	8	Description
%rax	%eax	%ax	%al	return value
%rbx	%ebx	%bx	%bl	Callee saved
%rcx	%ecx	%cx	%cl	4 th argument
%rdx	%edx	%dx	%dl	3 rd argument
%rsi	%esi	%si	%sil	2 nd argument
%rdi	%edi	%di	%dil	1 st argument
%rbp	%ebp	%bp	%bpl	Callee saved
%rsp	%esp	%sp	%spl	Stack pointer
%r8	%r8d	%r8w	%r8b	5 th argument
%r9	%r9d	%r9w	%r9b	6 th argument



System call

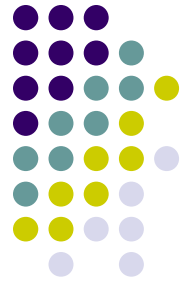
- Each system call has different arguments
- Assign parameters to appropriate registers
- Use int 0x80
- Example

Exit from the program

```
mov $0, %ebx  
mov $1, %eax  
int $0x80
```

Print a string

```
msg: .asciz "Hello World"  
mov $4, %eax  
mov $1, %ebx  
mov $msg, %ecx  
mov $10, %edx  
int $0x80
```

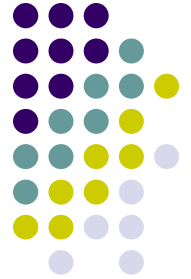


Call C library function

- 64bit architecture: use registers to pass arguments

```
.section .data
format_string: .asciz "Vendor ID: %d\n"
vendor_id: .int 12
.section .text
.globl _start
_start:
#Arguments to C functions:
mov $format_string, %rdi
mov vendor_id, %esi
mov $0, %eax
call printf
call exit
```

```
printf("Vendor ID is: %d\n",id);
```



Development tools

- compiler: [as](#), linker: [ld](#), debugger: [gdb](#)

```
.section .data
output: .asciz "The Vendor ID is '%d'\n"
vendor_id : .byte 12
.section .text
.globl _start
_start:
mov $format_string, %edi
mov vendor_id, %esi
mov $0, %eax
call printf
call exit
```

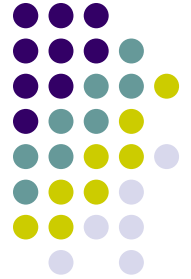
Compile, link and run the program

```
$ as -o print.o printf.s
```

```
$ ld -dynamic-linker /lib64/ld-linux-x86-64.so.2 -lc -o print print.o
```

```
$ ./print
```

Numeric types and conversions (cont'd)

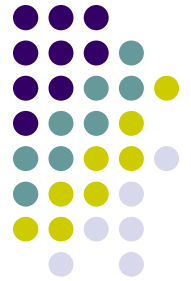


- Celcius to fahrenheit

```
double cel2fahr(float temp)
{
    return 1.8 * temp + 32;
}
```

convert the above function into an assembly procedure

Numeric types and conversions (cont'd)



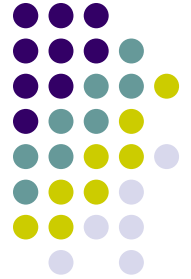
- Celcius to fahrenheit

#temp is in xmm0; scale: .double 1.8

proc_cel2fahrenheit:

mov \$32,%eax	#eax=32
cvtsi2sd %eax,%xmm2	#xmm2=double(eax)
movsd scale,%xmm1	#xmm1=scale
cvtss2sd %xmm0,%xmm0	#xmm0=double(xmm0)
mulsd %xmm1,%xmm0	#xmm0*=xmm1
addsd %xmm2,%xmm0	#xmm0+=xmm2
ret	

Numeric types and conversions (cont'd)



- Celcius to fahrenheit

```
double cel2fahr(int *temp) {  
    return 1.8 * (*temp) + 32.0;  
}
```

convert the above function into an assembly
procedure



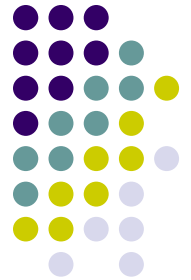
Numeric types and conversions (cont'd)

- Celcius to fahrenheit

#rdi=&temp

proc_cel2fahrenheit:

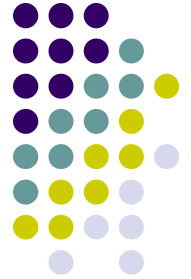
```
    mov 0(%rdi),%ebx      #ebx=*rdi;#ebx=temp
    cvtsi2sd %ebx,%xmm0   #xmm0=double(ebx)
    mov $32,%eax          #eax=32
    cvtsi2sd %eax,%xmm2   #xmm2=double(eax)
    movsd scale,%xmm1     #xmm1=scale
    mulsd %xmm1,%xmm0     #xmm0*=xmm1
    addsd %xmm2,%xmm0     #xmm0+=xmm2
    ret
```

Exercises

```
void proc(int a1, double *a1p)
{
    *a1p = a1*2.5;
}
```

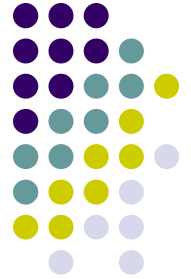
Convert the above function into an assembly procedure



Exercises (cont'd)

```
double fcvt(int i, float *fp, double *dp, long *lp)
{
    float f = *fp; double d = *dp; long l = *lp;
    *lp = (long) d;
    *fp = (float) i;
    *dp = (double) l;
    return (double) f;
}
```

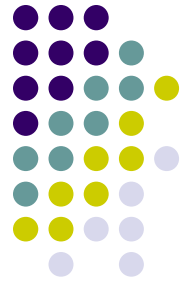
Convert the above function into an assembly procedure



Exercises (cont'd)

```
double funct(double a,  
             float x, double b, int i)  
{  
    return a*x - b/i;  
}
```

Convert the above function into an
assembly procedure



Exercises

- Write a procedure to print a number (in %eax)
- Write a program to print the value of factorial N ($N!$)
- Write a program to print the value of factorial N ($N!$) in a recursive procedure
- Write a program to print the product of two integer numbers ($a*b$) by an addition procedure
- Write a program to print the dividend of two integer numbers ($a\%b$) by a recursive subtraction procedure
- Write a program to calculate the sum of an array
- Write a program to calculate the sum of the first n natural numbers ($1+2+3+\dots+n$)



Exercises cont'd

- Write a program to print the first n fibonacci numbers
- Write a program to print the first of n numbers of a *geometric sequence* with a given value of a and r
- Write a program to print the first of n number in an *arithmetic sequence* with a given value of d and u
- Write a program to find out the greatest common divisor of the two numbers a and b
- Write a program to find out the *lowest common multiple* of the two numbers a and b
- Write a program to sort an array



Exercises cont'd

Fast calculate the function

$f(x) = a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \dots + a_1 x + a_0$ with the following method

$$f(x) = (a_n x + a_{n-1})x + a_{n-2})x + \dots + a_1)x + a_0$$

Where a_i is the element of an array float

$a[n+1]$. For example: float $a[] = \{1, 2, 3, 4, 5\}$;

then $a_0 = 1, a_1 = 2, \dots, a_n = 5$



Fibonacci

```
ebx=1; eax=1;  
for (ecx=3; ecx<=n; ecx++) {  
    r8d=ebx+eax;  
    ebx=eax;  
    eax=r8d;  
}
```



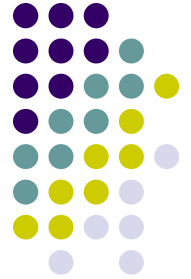
Fibonacci-recursive version

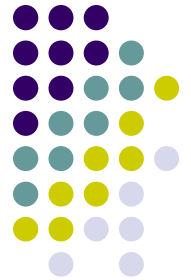
```
unsigned long fibonacci(unsigned long
n) {
    if(n<=2) return 1;
    n1=fibonacci(n-1);
    n2=fibonacci(n-2);
    return n1+n2;
}
```


Factorial

```
eax=1;
```

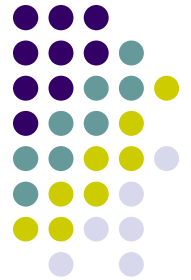
```
for (ebx=1 ; ebx<=n ; ebx++) eax*=ebx;
```





Factorial-recursive version

```
unsigned long fact(unsigned long n) {  
    if(n==1) return 1;  
    unsigned long t=fact(n-1);  
    t*=n;  
    return t;  
}
```



Geometric sequence

```
#a (n) = a (n-1) * r = a . rn-1;
```

```
xmm0 = a ;
```

```
xmm1 = r ;
```

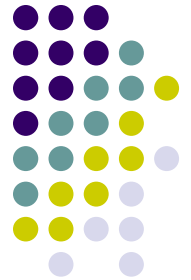
```
for (ecx=0 ; ecx<n ; ecx++) xmm0 *= xmm1 ;
```

Equation $ax+b=0$



```
xmm0=a; xmm2=b; xmm1=0;
if (xmm0==xmm1) {    #a==0?
    if (xmm2!=xmm1)    #b==0
        edx=-1; #impossible equation
    else edx=0;    #countless solution
}else{
    edx=1; #one solution
    xmm0=-xmm2/xmm0;
}
```

Maximum number of an array

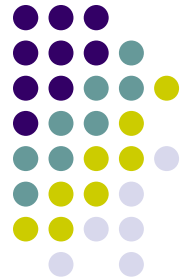


```
xmm0=a[0];  
for (ecx=1;ecx<n;ecx++)  
    if (xmm0<a[ecx]) xmm0=a[ecx];
```

Sum of an array

```
xmm0=0;  
for (ecx=0;ecx<n;ecx++)  
    xmm0+=a[ecx];
```

Equation $ax^2+bx+c=0$ ($a \neq 0$)

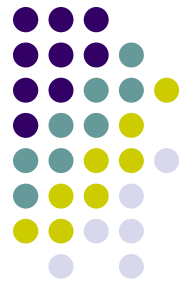
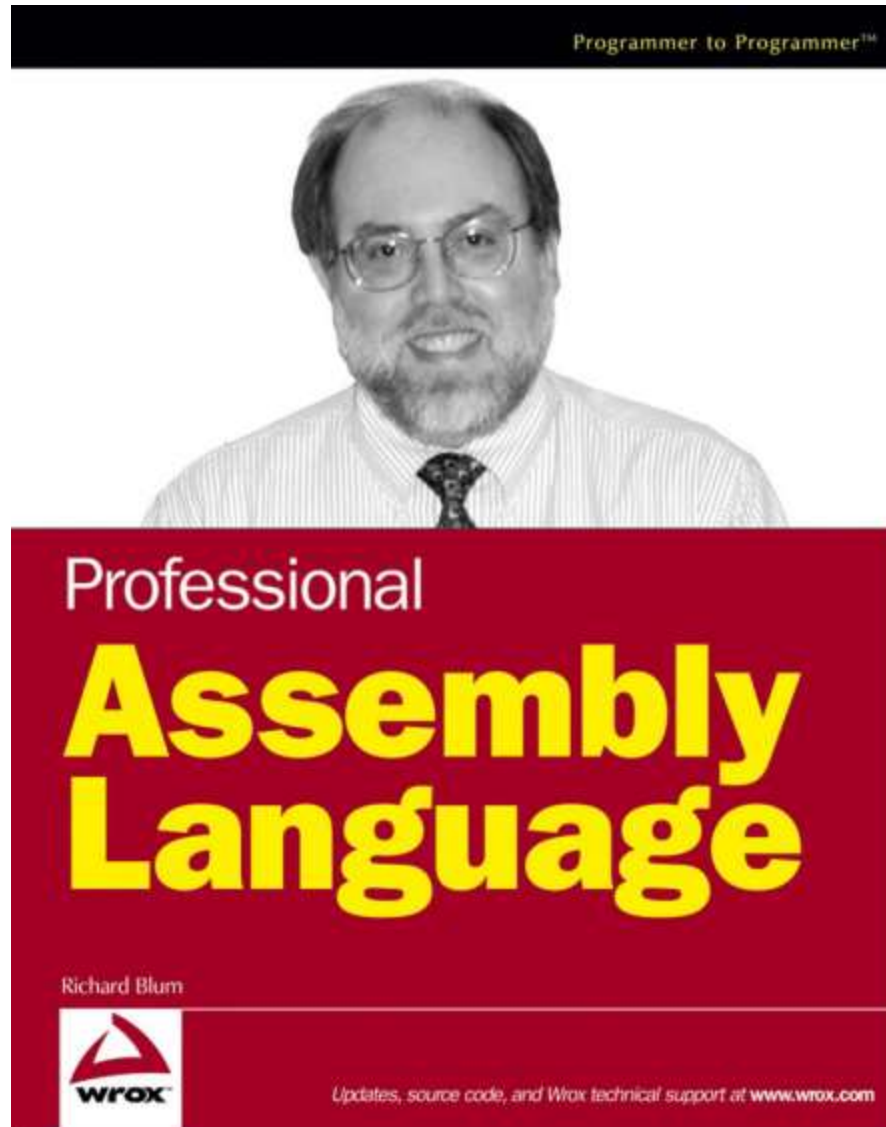


```
xmm5=a; xmm1=b; xmm2=c; xmm3=0;
xmm4=xmm1*xmm1-4*xmm5*xmm2; #delta=b*b-4*a*c;
if(xmm4<xmm3) edx=0; #impossible equation
else if(xmm4==xmm3){
    edx=1; xmm0=-xmm1/xmm5; #one solution
}else {
    edx=2; #two solutions
    xmm0=(-xmm1-sqrt(xmm4))/(2*xmm5);
    xmm1=(-xmm1+sqrt(xmm4))/(2*xmm5);
}
```

Reference

- Professional Assembly Language,

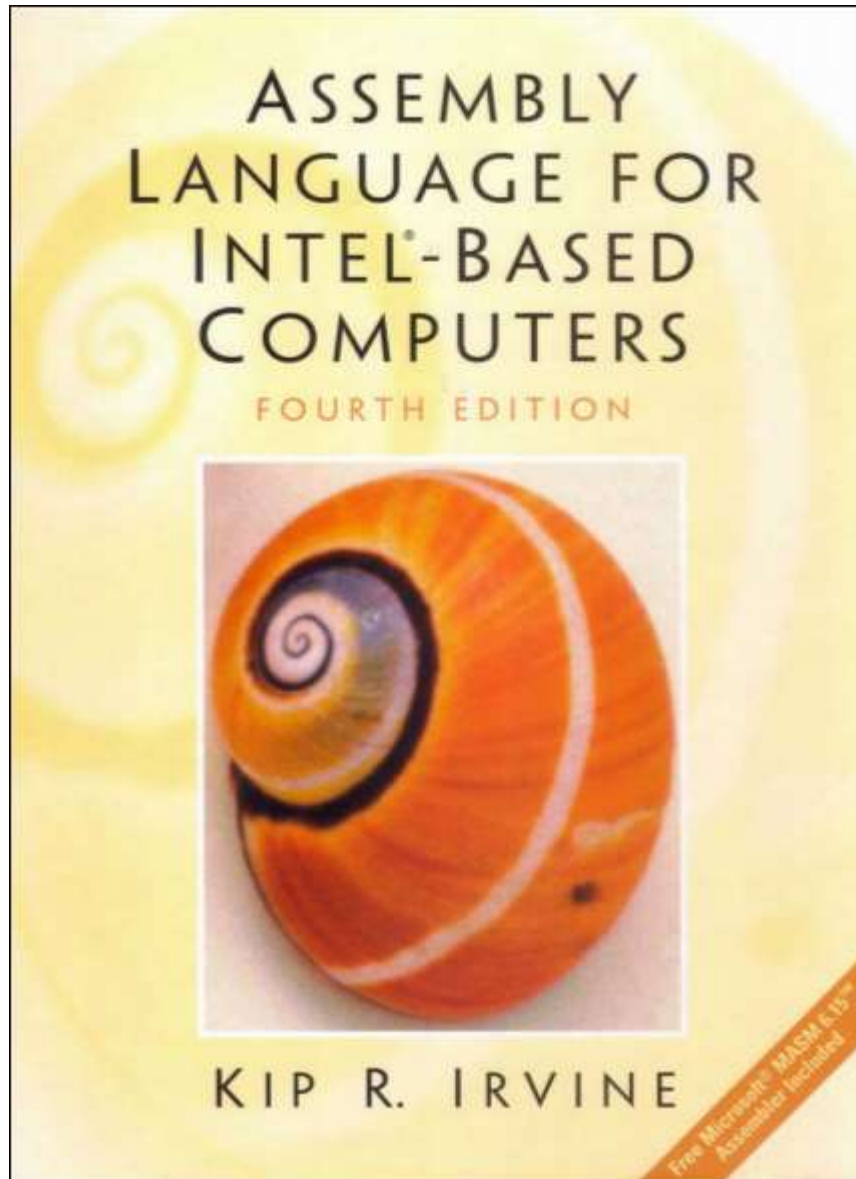
Richard Blum,
2005



Reference

- Assembly Language for Intel-Based Computers,

Kip R.Irvine, 2003



Reference

<http://x86.renejeschke.de/>

https://en.wikipedia.org/wiki/X86_instruction_listings

Intel® 64 and IA-32
Architectures
Software Developer's
Manual
Volume 2 (2A, 2B & 2C):
Instruction Set Reference,
A-Z

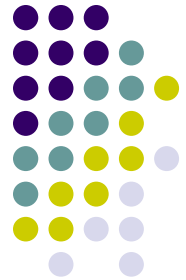


AMD64 Technology

AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions

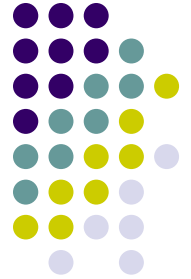
Publication No.	Revision	Date
24594	3.15	November 2009

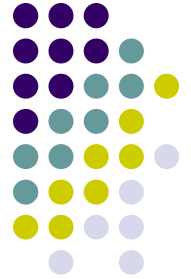
Advanced Micro Devices 



End of chapter

- Happy coding!
- Any questions?





Intel-based Assembly

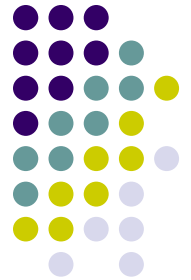
APPENDICES

Appendix: Call C library function in 32 bit architecture



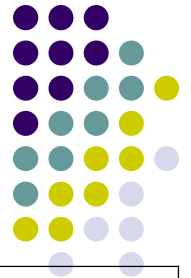
- Use stack to pass arguments

```
.section .data
output: .asciz "The Vendor ID is '%d'\n"
buffer: .byte 12
.section .text
.globl _start
_start:
push $12
push $output
call printf
addl $8, %esp
push $0
call exit
```



x86 FP Architecture

- Originally based on 8087 FP coprocessor
 - 8 × 80-bit extended-precision registers
 - Used as a push-down stack
 - Registers indexed from TOS: ST(0), ST(1), ...
- FP values are 32-bit or 64 in memory
 - Converted on load/store of memory operand
 - Integer operands can also be converted on load/store
- Very difficult to generate and optimize code
 - Result: poor FP performance



x86 FP Instructions

Data transfer	Arithmetic	Compare	Transcendental
FILD mem/ST(i) FISTP mem/ST(i) FLDPI FLD1 FLDZ	FIADDP mem/ST(i) FISUBRP mem/ST(i) FIMULP mem/ST(i) FIDIVRP mem/ST(i) FSQRT FABS FRNDINT	FICOMP FIUCOMP FSTSW AX/mem	FPATAN F2XMI FCOS FPTAN FPREM FPSIN FYL2X

- Optional variations
 - I: integer operand
 - P: pop operand from stack
 - R: reverse operand order
 - But not all combinations allowed