

## Bibliografia

Questo capitolo contiene l'elenco alfabetico di tutti i libri e gli articoli citati nel volume.

- Abramson, D., "Mixing Cloud and Grid Resources for Many Task Computing", *Atti dell'Int'l Workshop on Many Task Computing on Grids and Supercomputers*, ACM, pp. 1–2, 2011.
- Adams, M. e Dulchinos, D., "OpenCable", *IEEE Commun. Magazine*, vol. 39, pp. 98–05, giugno 2001.
- Adiga, N.R. et al., "An overview of the BlueGene/L Supercomputer", *Atti di Supercomputing 2002*, ACM, pp. 1–22, 2002.
- Adve, S.V. e Hill, M., "Weak Ordering: a new definition", *Atti del 17<sup>a</sup> Ann. Int'l Symp. On Computer Arch.*, ACM, pp. 2–14, 1990.
- Agerwala, T. e Cocke, J., "High performance reduced instruction set processors", *IBM T.J. Watson Research Center Technical Report RC12434*, 1987.
- Ahmadinia, A. e Shahrary, A., "A Highly Adaptive and Efficient Router Architecture for Network-on-Chip", *Computer J.*, vol. 54, pp. 1295–1307, agosto 2011.
- Alam, S., Barrett, R., Bast, M., Fahey, M.R., Kuehn, J., McCurdy, Rogers, J., Roth, P., Sankaran, R., Vetter, J.S., Worley, P. e Yu, W., "Early Evaluation of IBM BlueGene/P", *Atti dell'ACM/IEEE Conference on Supercomputing*, ACM/IEEE, 2008.
- Alameldeen, A.R. e Wood, D.A., "Adaptive cache compression for high-performance processors", *Atti del 31<sup>a</sup> Ann. Int'l Sym. on Computer Arch.*, ACM, pp. 212–223, 2004.
- Almasi, G.S. et al., "System management in the BlueGene/L Supercomputer", *Atti del 17<sup>o</sup> Int'l Parallel and Distr. Symp.*, IEEE, 2003a.
- Almasi, G.S. et al., "An overview of the Bluegene/L System Software Organization", *Par. Proc. Letters*, vol. 13, pp. 561–574, aprile 2003b.

- Amza, C., Cox, A., Dwarkadas, S., Keleher, P., Lu, H., Rajamony, R., Yu, W. e Zwaenepoel, W., "TreadMarks: shared memory computing on a network of workstations", *IEEE Computer Magazine*, vol. 29, pp. 18–28, febbraio 1996.
- Anderson, D., *Universal serial bus system architecture*, Reading, MA, Addison-Wesley, 1997.
- Anderson, D., Budruk, R., e Shanley, T., *PCI express system architecture*, Reading, MA, Addison-Wesley, 2004.
- Anderson, T.E., Culler, D.E. e Patterson D.A., "A case for NOW (Networks of workstations)", *IEEE Micro Magazine*, vol. 15, pp. 54–64, gennaio 1995.
- August, D.I., Connors, D.A., Mahlke, S.A., Sias, J.W., Crozier, K.M., Cheng, B.-C., Eaton, P.R., Olaniran, Q.B. e Hwu, W.-M., "Integrated predicated and speculative execution in the IMPACT EPIC Architecture", *Atti del 25º Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 227–237, 1998.
- Bal, H.E., *Programming distributed systems*, Hemel Hempstead, England, Prentice Hall Int'l, 1991.
- Bal, H.E., Bhoedjang, R., Hofman, R., Jacobs, C., Langendoen, K., Ruhl, T. e Kaashoek, M.F., "Performance evaluation of the Orca Shared Object System", *ACM Trans. on Computer Systems*, vol. 16, pp. 1–40, gennaio-febbraio 1998.
- Bal, H.E., Kaashoek, M.F. e Tanenbaum, A.S., "Orca: a language for parallel programming of distributed systems", *IEEE Trans. on Software Engineering*, vol. 18, pp. 190–205, marzo 1992.
- Bal, H.E. e Tanenbaum, A.S., "Distributed programming with shared data", *Atti della 1988 Int'l Conf. on Computer Languages*, IEEE, pp. 82–91, 1988.
- Balasangameshwara, J. e Raju, N., "A Hybrid Policy for Fault Tolerant Load Balancing in Grid Computing Environments", *J. Network and Computer Applications*, vol. 35, pp. 412–422, gennaio 2012.
- Barroso, L.A., Dean, J., e Holzle, U., "Web search for a planet: the Google Cluster Architecture", *IEEE Micro Magazine*, vol. 23, pp. 22–28, marzo-aprile 2003.
- Bechini, A., Conte, T.M. e Prete, C.A., "Opportunities and challenges in embedded systems", *IEEE Micro Magazine*, vol. 24, pp. 8–9, luglio-agosto 2004.
- Bhakthavatchalu, R., Deepthy, G.R. e Shanoja, S., "Implementation of Reconfigurable Open Core Protocol Compliant Memory System Using VHDL", *Atti dell'Int Conf. on Industrial and Information Systems*, pp. 213–218, 2010.
- Bjornson, R.D., "Linda on distributed memory multiprocessors", tesi di dottorato, Yale Univ., 1993.
- Blumrich, M., Chen, D., Chiu, G., Coteus, P., Gara, A., Giampaapa, M.E., Haring, R.A., Heidelberger, P., Hoenicke, D., Kopesay, G.V., Ohmacht, M., Steinmacher-Burow, B.D., Takken, T., Vransas, P. e Liebsch, T., "An overview of the BlueGene/L System", *IBM J. Research and Devel.*, vol. 49, marzo-maggio 2005.
- Bose, P., "Computer architecture research: shifting priorities and newer challenger", *IEEE Micro Magazine*, vol. 24, p. 5, novembre-dicembre 2004.
- Bouknight, W.J., Denenberg, S.A., McIntyre, D.E., Randall, J.M., Sameh, A.H. e Slotnick, D.L., "The ILLIAC IV System", *Atti IEEE*, pp. 369–388, aprile 1972.
- Bradley, D., "A Personal History of the IBM PC", *IEEE Computer*, vol. 44, pp. 19–25, agosto 2011.
- Bride, E., "The IBM Personal Computer: A Software-Driven Market", *IEEE Computer*, vol. 44, pp. 34–39, agosto 2011.
- Brightwell, R., Camp, W., Cole, B., DeBenedictis, E., Leland, R., Tompkins, H. e MacCabe, A.B., "Architectural specification for massively parallel computers. An experience and measurement-based approach", *Concurrency and computation: practice and experience*, vol. 17, pp. 1–46, 2005.
- Brightwell, R., Underwood, K.D., Vaughan, C. e Stevenson, J., "Performance Evaluation of the Red Storm Dual-Core Upgrade", *Concurrency and Computation: Practice and Experience*, vol. 22, pp. 175–190, febbraio 2010.
- Burkhardt, H., Frank, S., Knobe B., e Rothnie, J., "Overview of the KSR-1 Computer System", *Technical Report KSR-TR-9202001*, Kendall Square Research Corp, Cambridge, MA, 1992.
- Carrieri, N. e Gelernter, D., "Linda in context", *Comunicazioni dell'ACM*, vol. 32, pp. 444–458, aprile 1989.
- Celaya, J. e Arronategui, U., "A Highly Scalable Decentralized Scheduler of Tasks with Deadlines", *Atti della 12ª Int'l Conf. on Grid Computing*, IEEE/ACM, pp. 58–65, 2011.
- Charlesworth, A., "The Sun fireplane interconnect", *IEEE Micro Magazine*, vol. 22, pp. 36–45, gennaio-febbraio 2002.
- Charlesworth, A., "The Sun fireplane interconnect", *Atti della Conf. on High Perf Networking and Computing*, ACM, 2001.
- Chen, L., Dropsho, S. e Albonesi, D.H., "Dynamic data dependence tracking and its application to branch prediction", *Atti del 9º Int'l Symp. on High-Performance Computer Arch.*, IEEE, pp. 65–78, 2003.
- Cheng, L. e Carter, J.B., "Extending CC-NUMA Systems to Support Write Update Optimizations", *Atti della 2008 ACM/IEEE Conf. on Supercomputing*, ACM/IEEE, 2008.
- Chou, Y., Fahs, B. e Abraham, S., "Microarchitecture optimizations for exploiting Memory-level parallelism", *Atti del 31º Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 76–77, 2004.
- Cohen, D., "On holy wars and a plea for peace", *IEEE Computer Magazine*, vol. 14, pp. 48–54, ottobre 1981.
- Corbato, F.J. e Vyssotsky, V.A., "Introduction and overview of the MULTICS System", *Atti FJCC*, pp. 185–196, 1965.
- Denning, P.J., "The working set model for program behavior", *Comunicazioni dell'ACM*, vol. 11, pp. 323–333, maggio 1968.
- Dijkstra, E.W., "GOTO statement considered harmful", *Comunicazioni dell'ACM*, vol. 11, pp. 147–148, marzo 1968a.
- Dijkstra, E.W., "Co-operating sequential processes", *Programming Languages*, F. Genuys (a cura di), New York, Academic Press, 1968b.
- Donaldson, G. e Jones, D., "Cable television broadband network architectures", *IEEE Commun. Magazine*, vol. 39, pp. 122–126, giugno 2001.

- Dubois, M., Scheurich, C. e Briggs, F.A., "Memory access buffering in multiprocessors", *Atti del 13<sup>o</sup> Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 434–442, 1986.
- Dulong, C., "The IA-64 architecture at work", *IEEE Computer Magazine*, vol. 31, pp. 24–32, luglio 1998.
- Dutta-Roy, A., "An overview of cable modem technology and market perspectives", *IEEE Commun. Magazine*, vol. 39, pp. 81–88, giugno 2001.
- Faggin, F., Hoff, M.E., Mazor, S., Jr. e Shima, M., "The history of the 4004", *IEEE Micro Magazine*, vol. 16, pp. 10–20, novembre 1996.
- Falcon, A., Stark, J., Ramirez, A., Lai, K. e Valero, M., "Prophet/critic hybrid branch prediction", *Atti del 31<sup>o</sup> Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 250–261, 2004.
- Fisher, J.A. e Freudenberger, S.M., "Predicting conditional branch directions from previous runs of a program", *Atti della 5<sup>a</sup> Int'l Conf. on Arch. Support for Prog. Lang. and Operating Syst.*, ACM, pp. 85–95, 1992.
- Flynn, D., "AMBA: enabling reusable on-chip designs", *IEEE Micro Magazine*, vol. 17, pp. 20–27, luglio 1997.
- Flynn, M.J., "Some computer organizations and their effectiveness", *IEEE Trans. On Computers*, vol. C-21, pp. 948–960, settembre 1972.
- Foster, I. e Kesselman, C., *The grid 2: blueprint for a new computing infrastructure*, San Francisco, Morgan Kaufman, 2003.
- Fotheringham, J., "Dynamic storage allocation in the atlas computer including an automatic use of a backing store", *Comunicazioni dell'ACM*, vol. 4, pp. 435–436, ottobre 1961.
- Freitas, H.C., Madruga, F.L., Alves, M. e Navaux, P., "Design of Interleaved Multithreading for Network Processors on Chip", *Atti dell'Int. Symp. on Circuits and Systems*, IEEE, 2009.
- Gaspar, L., Fischer, V., Bernard, F., Bossuet, L.-e Cotret, P., "HCrypt: A Novel Concept of Crypto-processor with Secured Key Management", *Int. Conf. on Reconfigurable Computing and FPGAs*, 2010.
- Gaur, J., Chaudhuri, C. e Subramoney, S., "Bypass and Insertion Algorithms for Exclusive Last-level Caches", *Atti del 38<sup>o</sup> Int. Symp. on Computer Arch.*, ACM, 2011.
- Gebhart, M., Johnson, D.R., Tarjan, D., Keckler, S.W., Dally, W.J., Lindholm, E. e Skadron, K., "Energy-efficient Mechanisms for Managing Thread Context in Throughput Processors", *Atti del 38<sup>o</sup> Int. Symp. on Computer Arch.*, ACM, 2011.
- Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Mancheck, R. e Sunderram, V., *PVM: Parallel Virtual Machine – A user's guide and tutorial for networked parallel computing*, Cambridge, MA, MIT Press, 1994.
- Gepner, P., Gamayunov, V. e Fraser, D.L., "The 2nd Generation Intel Core Processor. Architectural Features Supporting HPC", *Atti del 10<sup>o</sup> Int. Symp. on Parallel and Dist. Computing*, pp. 17–24, 2011.
- Gerber, R. e Binstock, A., *Programming with hyper-threading technology*, Santa Clara, CA, Intel Press, 2004.
- Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P.B., Gupta, A. e Hennessy, J.L., "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors", *Atti del 17<sup>o</sup> Ann. Int. Symp. on Comp. Arch.*, ACM, pp. 15–26, 1990.
- Ghemawat, S., Gobioff, H. e Leung, S.-T., "The google file system", *Atti del 19<sup>o</sup> Symp. on Operating Systems Principles*, ACM, pp. 29–43, 2003.
- Goodman, J.R., "Using cache memory to reduce processor memory traffic", *Atti del 10<sup>o</sup> Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 124–131, 1983.
- Goodman, J.R., "Cache consistency and sequential consistency", *Tech. Rep. 61*, IEEE Scalable Coherent Interface Working Group, IEEE, 1989.
- Goth, G., "IBM PC Retrospective: There Was Enough Right to Make It Work", *IEEE Computer*, vol. 44, pp. 26–33, agosto 2011.
- Gropp, W., Lusk, E. e Skjellum, A., *Using MPI: portable parallel programming with the message passing interface*, Cambridge, MA, MIT Press, 1994.
- Gupta, N., Mandal, S., Malave, J., Mandal, A. e Mahapatra, R.N., "A Hardware Scheduler for Real Time Multiprocessor System on Chip", *Atti della 23<sup>a</sup> Int. Conf. on VLSI Design*, IEEE, 2010.
- Gurumurthi, S., Sivasubramaniam, M., Kandemir, M. e Franke H., "Reducing disk power consumption in servers with DRPM", *IEEE Computer Magazine*, vol. 36, pp. 59–66, dicembre 2003.
- Hagersten, E., Landin, A. e Haridi, S., "DDM – A cache-only memory architecture", *IEEE Computer Magazine*, vol. 25, pp. 44–54, settembre 1992.
- Haghaghizadeh, F., Attarzadeh, H. e Sharifkhani, M., "A Compact 8-Bit AES Crypto-processor", *Atti della 2<sup>a</sup> Int. Conf. on Computer and Network Tech.*, IEEE, 2010.
- Hamming, R.W., "Error detecting and error correcting codes", *Bell Syst. Tech. J.*, vol. 29, pp. 147–160, aprile 1950.
- Henkel, J., Hu, X.S. e Bhattacharyya, S.S., "Taking on the embedded system challenge", *IEEE Computer Magazine*, vol. 36, pp. 35–37, aprile 2003.
- Hennessy, J.L., "VLSI processor architecture", *IEEE Trans. on Computers*, vol. C-33, pp. 1221–1246, dicembre 1984.
- Herrero, E., Gonzalez, J. e Canal, R., "Elastic Cooperative Caching: An Autonomous Dynamically Adaptive Memory Hierarchy for Chip Multiprocessors", *Atti della 23<sup>a</sup> Int. Conf. on VLSI Design*, IEEE, 2010.
- Hoare, C.A.R., "Monitors, an operating system structuring concept", *Comunicazioni dell'ACM*, vol. 17, pp. 549–557, ottobre 1974 (*erratum in Comunicazioni dell'ACM*, vol. 18, p. 95, febbraio 1975).
- Hwu, W.-M., "Introduction to predicated execution", *IEEE Computer Magazine*, vol. 31, pp. 49–50, gennaio 1998.
- Jimenez, D.A., "Fast path-based neural branch prediction", *Atti del 36<sup>o</sup> Int. Symp. on Microarchitecture*, IEEE, pp. 243–252, 2003.
- Johnson, K.L., Kaashoek, M.F. e Wallach, D.A., "CRL: high-performance all-software distributed shared memory", *Atti del 15<sup>o</sup> Symp. on Operating Systems Principles*, ACM, pp. 213–228, 1995.

- Kapasi, U.J., Rixner, S., Dally, W.J., Khailany, B., Ahn, J.H., Mattson, P. e Owens, J.D., "Programmable stream processors", *IEEE Computer Magazine*, vol. 36, pp. 54–62, agosto 2003.
- Kaufman, C., Perlman, R. e Speciner, M., *Network Security*, seconda edizione, Upper Saddle River, NJ, Prentice Hall, 2002.
- Kim, N.S., Austin, T., Blaauw, D., Mudge, T., Flautner, K., Hu, J.S., Irwin, M.J., Kandemir, M. e Narayanan, V., "Leakage current: Moore's Law meets static power", *IEEE Computer Magazine*, vol. 36, pp. 68–75, dicembre 2003.
- Knuth, D.E., *The art of computer programming: fundamental algorithms*, terza edizione, Reading, MA, Addison-Wesley, 1997.
- Kontothanassis, L., Hunt, G., Stets, R., Hardavellas, N., Cierniad, M., Parthasarathy, S., Meira, W., Dwarkadas, S. e Scott, M., "VM-based shared memory on low latency remote memory access networks", *Atti del 24º Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 157–169, 1997.
- Lamport, L., "How to make a multiprocessor computer that correctly executes multiprocess programs", *IEEE Trans. on Computers*, vol. C-28, pp. 690–691, settembre 1979.
- Larowe, R.P. ed Ellis C.S., "Experimental comparison of memory management policies for NUMA multiprocessors", *ACM Trans. on Computer Systems*, vol. 9, pp. 319–363, novembre 1991.
- Lee, J., Keleher, P. e Sussman, A., "Supporting Computing Element Heterogeneity in P2P Grids", *Atti dell'IEEE Int. Conf. on Cluster Computing*, IEEE, pp. 150–158, 2011.
- Li, K. e Hudak, P., "Memory coherence in shared virtual memory systems", *ACM Trans. On Computer Systems*, vol. 7, pp. 321–359, novembre 1989.
- Lin, Y.-N., Lin, Y.-D. e Lai, Y.-C., "Thread Allocation in CMP-based Multithreaded Network Processors", *Parallel Computing*, vol. 36, pp. 104–116, febbraio 2010.
- Lu, H., Cox, A.L., Dwarkadas S., Rajamony, R. e Zwaenepoel, W., "Software distributed shared memory support for irregular applications", *Atti della 6ª Conf. on Prin. and Practice of Parallel Progr.*, pp. 48–56, giugno 1997.
- Lukasiewicz, J., *Aristotle's syllogistic*, seconda edizione, Oxford, Oxford University Press, 1958.
- Lyytinen, K. e Yoo, Y., "Issues and challenges in ubiquitous computing", *Comunicazioni dell'ACM*, vol. 45, pp. 63–65, dicembre 2002.
- Martin, R.P., Vahdat, A.M., Culler, A.M. e Anderson, T.E., "Effects of communication latency, overhead and bandwidth in a cluster architecture", *Atti del 24º Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 85–97, 1997.
- Mayhew, D. e Krishnan, V., "PCI express and advanced switching: evolutionary path to building next generation interconnects", *Atti dell'11º Symp. on High Perf. Interconnects* IEEE, pp. 21–29, agosto 2003.
- McKusick, M.K., Joy, W.N., Leffler, S.J. e Fabry, R.S., "A fast file system for UNIX", *ACM Trans. on Computer Systems*, vol. 2, pp. 181–197, agosto 1984.
- McNairy, C. e Soltis, D., "Itanium 2 processor microarchitecture", *IEEE Micro Magazine*, vol. 23, pp. 44–55, marzo-aprile 2003.
- Mishra, A.K., Vijaykrishnan, N. e Das, C.R., "A Case for Heterogeneous On-Chip Interconnects for CMPs", *Atti del 38º Int'l Symp. on Computer Arch.*, ACM, 2011.
- Morgan, C., *Portraits in Computing*, New York, ACM Press, 1997.
- Moudgil, M. e Vassiliadis, S., "Precise interruptus", *IEEE Micro Magazine*, vol. 16, pp. 58–67, gennaio 1996.
- Mullender, S.J. e Tanenbaum, A.S., "Immediate files", *Software-Practice and Experience*, vol. 14, pp. 365–368, 1984.
- Naeem, A., Chen, X., Lu, Z. e Jantsch, A., "Realization and Performance Comparison of Sequential and Weak Memory Consistency Models in Network-On-Chip Based Multicore Systems", *Atti della 16ª Design Automation Conf. Asia and South Pacific*, IEEE, pp. 154–159, 2011.
- Organick, E., *The MULTICS system*, Cambridge, MA, MIT Press, 1972.
- Oskin, M., Chong, F.T. e Chuang, I.L., "A practical architecture for reliable quantum computers", *IEEE Computer Magazine*, vol. 35, pp. 79–87, gennaio 2002.
- Papamarcos, M. e Patel, J., "A low overhead coherence solution for multiprocessors with private cache memories", *Atti dell'11º Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 348–354, 1984.
- Parikh, D., Skadron, K., Zhang, Y. e Stan, M., "Power-Aware Branch Prediction: characterization and design", *IEEE Trans. on Computers*, vol. 53, pp. 168–186, febbraio 2004.
- Patterson, D.A., "Reduced instruction set computers", *Comunicazioni dell'ACM*, vol. 28, pp. 8–21, gennaio 1985.
- Patterson, D.A., Gibson, G. e Katz, R., "A case for redundant arrays of inexpensive disks (RAID)", *Atti dell'ACM SIGMOD Int'l Conf. on Management of Data*, ACM, pp. 109–166, 1988.
- Patterson, D.A. e Sequin, C.H., "A VLSI RISC", *IEEE Computer Magazine*, vol. 15, pp. 8–22, settembre 1982.
- Pountain, D., "Pentium: more RISC than CISC", *Byte*, vol. 18, pp. 195–204, settembre 1993.
- Radin, G., "The 801 minicomputer", *Computer Arch. News*, vol. 10, pp. 39–47, marzo 1982.
- Raman, S.K., Pentkovski, V. e Keshava, J., "Implementing streaming SIMD extensions on the Pentium III Processor", *IEEE Micro Magazine*, vol. 20, pp. 47–57, luglio-agosto 2000.
- Ritchie, D.M., "Reflections on Software Research", *Comunicazioni dell'ACM*, vol. 27, pp. 758–760, agosto 1984.
- Ritchie, D.M. e Thompson, K., "The UNIX time-sharing system", *Comunicazioni dell'ACM*, vol. 17, pp. 365–375, luglio 1974.
- Robinson, G.S., "Toward the age of smarter storage", *IEEE Computer Magazine*, vol. 35, pp. 35–41, dicembre 2002.
- Rosenblum, M. e Ousterhout, J.K., "The design and implementation of a log-structured file system", *Atti del 13º Symp. on Operating System Principles*, ACM, pp. 1–15, 1991.

- Russinovich, M.E. e Solomon, D.A., *Microsoft Windows internals*, quarta edizione, Redmond, WA, Microsoft Press, 2005.
- Rusu, S., Muljono, H. e Cherkauer, B., "Itanium 2 Processor 6M", *IEEE Micro Magazine*, vol. 24, pp. 10–18, marzo-aprile 2004.
- Saha, D. e Mukherjee, A., "Pervasive computing: a paradigm for the 21st century", *IEEE Computer Magazine*, vol. 36, pp. 25–31, marzo 2003.
- Sakamura, K., "Making computers invisible", *IEEE Micro Magazine*, vol. 22, p. 711, 2002.
- Sanchez, D. e Kozyrakis, C., "Vantage: Scalable and Efficient Fine-Grain Cache Partitioning", *Atti del 38º Ann. Int. Symp. on Computer Arch.*, ACM, pp. 57–68, 2011.
- Scales, D.J., Gharachorloo, K. e Thekkath, C.A., "Shasta: a low-overhead software-only approach for supporting fine-grain shared memory", *Atti della 7ª Int'l Conf. on Arch. Support for Prog. Lang. and Oper. Syst.*, ACM, pp. 174–185, 1996.
- Seltzer, M., Bostic, K., McKusick, M.K. e Staelin, C., "An implementation of a log-structured file system for UNIX", *Atti inverno 1993 USENIX Technical Conf.*, pp. 307–326, 1993.
- Shanley, T. e Anderson, D., *PCI system architecture*, quarta edizione, Reading, MA, Addison-Wesley, 1999.
- Shoufan, A., Huber, N. e Molter, H.G., "A Novel Cryptoprocessor Architecture for Chained Merkle Signature Schemes", *Microprocessors and Microsystems*, vol. 35, pp. 34–47, febbraio 2011.
- Singh, G.: "The IBM PC: The Silicon Story", *IEEE Computer*, vol. 44, pp. 40–45, agosto 2011.
- Slater, R., *Portraits in silicon*, Cambridge, MA, MIT Press, 1987.
- Snir, M., Otto, S.W., Huss-Lederman, S., Walker, D.W. e Dongarra, J., *MPI: the complete reference manual*, Cambridge, MA, MIT Press, 1996.
- Solari, E. e Congdon, B., *PCI express design & system architecture*, Research Tech, INC., 2005.
- Solari, E. e Willse, G., *PCI and PCI-X hardware and software*, sesta edizione, San Diego, CA, Annabooks, 2004.
- Sorin, D.J., Hill, M.D. e Wood, D.A., *A Primer on Memory Consistency and Cache Coherence*, San Francisco, Morgan & Claypool, 2011.
- Stets, R., Dwarkadas, S., Hardavellas, N., Hunt, G., Kontothanassis, L., Parthasarathy, S. e Scott, M., "CASHMERE-2L: software coherent shared memory on clustered remote-write networks", *Atti del 16º Symp. on Operating Systems Principles*, ACM, pp. 170–183, 1997.
- Summers, C.K., *ADSL: standards, implementation and architecture*, Boca Raton, FL, CRC Press, 1999.
- Sunderram, V.B., "PVM: a framework for parallel distributed computing", *Concurrency: practice and experience*, vol. 2, pp. 315–339, dicembre 1990.
- Swan, R.J., Fuller, S.H. e Siewiorek, D.P., "Cm\*-A modular multiprocessor", *Atti NCC*, pp. 645–655, 1977.
- Tan, W.M., *Developing USB PC Peripherals*, San Diego, CA, Annabooks, 1997.

- Tanenbaum, A.S. e Wetherall, D.J., *Computer Networks*, quinta edizione, Upper Saddle River, NJ, Prentice Hall, 2011.
- Thompson, K., "Reflections on Trusting Trust", *Comunicazioni dell'ACM*, vol. 27, pp. 761–763, agosto 1984.
- Thompson, J., Dreisigmeyer, D.W., Jones, T., Kirby, M. e Ladd, J., "Accurate Fault Prediction of BlueGene/P RAS Logs via Geometric Reduction", *IEEE*, pp. 8–14, 2010.
- Treleaven, P., "Control-Driven, Data-Driven, and Demand-Driven computer architecture", *Parallel Computing*, vol. 2, 1985.
- Tu, X., Fan, X., Jin, H., Zheng, L. e Peng, X., "Transactional Memory Consistency: A New Consistency Model for Distributed Transactional Memory", *Atti della 3ª Int. Joint Conf. on Computational Science and Optimization*, IEEE, 2010.
- Vahalia, U., *UNIX Internals*, Upper Saddle River, NJ, Prentice Hall, 1996.
- Vahid, F., "The softening of hardware", *IEEE Computer Magazine*, vol. 36, pp. 27–34, aprile 2003.
- Vetter, P., Goderis, D., Verpoorten L. e Granger, A., "Systems aspects of APON/VDSL deployment", *IEEE Commun. Magazine*, vol. 38, pp. 66–72, maggio 2000.
- Vu, T.D., Zhang, L. e Jesshope, C., "The Verification of the On-Chip COMA Cache Coherence Protocol", *Atti della 12ª Int'l Conf. on Algebraic Methodology and Software Technology*, Springer-Verlag, pp. 413–429, 2008.
- Weiser, M., "The computer for the 21st Century", *IEEE Pervasive Computing*, vol. 1, pp. 19–25, gennaio-marzo 2002 (pubblicato in originale su *Scientific American*, settembre 1991).
- Wilkes, M.V., "Computers then and now", *J. ACM*, vol. 15, pp. 1–7, gennaio 1968.
- Wilkes, M.V., "The best way to design an automatic calculating machine", *Atti della Manchester Univ. Computer Inaugural Conf.*, 1951.
- Wing-Kei, Y., Huang, R., Xu, S., Wang, S.-E., Kan, E. e Suh, G.E., "SRAM-DRAM Hybrid Memory with Applications to Efficient Register Files in Fine-Grained Multi-Threading Architectures", *Atti del 38º Int. Symp. on Computer Arch.*, ACM, 2011.
- Yamamoto, S. e Nakao, A., "Fast Path Performance of Packet Cache Router Using Multi-core Network Processor", *Atti del 7º Symp. on Arch. for Network and Comm. Sys.*, ACM/IEEE, 2011.
- Zhang, L. e Jesshope, C., "On-Chip COMA Cache-Coherence Protocol for Microgrids of Microthreaded Cores", *Atti della 2007 European Conf. on Parallel Processing*, Springer-Verlag, pp. 38–48, 2008.

L'aritmetica dei calcolatori è un po' diversa da quella che abbiamo studiato alle elementari. La differenza principale sta nel fatto che i calcolatori eseguono operazioni su numeri che hanno una precisione finita e prefissata. Un'altra differenza è che quasi tutti i calcolatori rappresentano i numeri nel sistema binario e non in quello decimale. Questa appendice si occupa precisamente di questi argomenti.

### A.1 Numeri a precisione finita

Quando si fanno calcoli aritmetici, in genere non si dà particolare importanza al numero di cifre decimali utilizzate per rappresentare i numeri. I fisici calcolano che nell'universo esistono  $10^{79}$  elettroni, senza preoccuparsi del fatto che ci vogliono 79 cifre decimali per scrivere quel numero per esteso. Una persona che usi carta e penna per calcolare il valore di una funzione e che abbia bisogno di una risposta con sei cifre significative, svolgerà i calcoli mantenendo i risultati intermedi con una precisione di sette, otto cifre, o di quante siano necessarie. Non succede mai che il foglio di carta sia troppo piccolo per contenere i numeri con sette cifre significative.

Le cose sono molto diverse nei calcolatori. Per molti di loro, la quantità di memoria disponibile per la memorizzazione di un numero è fissata al momento della progettazione. Il programmatore può riuscire con qualche sforzo a raddoppiare o a triplicare il numero di cifre che può usare, ma ciò non cambia la natura del problema. La natura intrinsecamente finita di un calcolatore ci costringe a trattare solo numeri rappresentati per mezzo di un numero limitato e costante di cifre, cioè **numeri a precisione finita**.

Per studiare le proprietà dei numeri a precisione finita, cominciamo con l'esaminare l'insieme degli interi positivi rappresentabili con tre cifre decimali, senza virgola e senza segno. Questo insieme ha esattamente 1000 elementi: 000, 001, 002, 003, ..., 999. Con queste limitazioni non possiamo esprimere:

1. i numeri più grandi di 999;
2. i numeri negativi;
3. le frazioni (proprie);
4. i numeri irrazionali;
5. i numeri complessi.

Una proprietà importante dell'aritmetica sull'insieme degli interi è la chiusura rispetto alle operazioni di addizione, sottrazione e moltiplicazione. In altre parole, per ogni coppia d'interi  $i$  e  $j$ , i risultati delle operazioni  $i + j$ ,  $i - j$  e  $i \times j$  sono tutti interi. L'insieme degli interi non è chiuso rispetto alla divisione, perché esistono valori di  $i$  e  $j$  per cui  $i/j$  non è esprimibile come un intero (per esempio  $7/2$  e  $1/0$ ).

I numeri a precisione finita non sono chiusi rispetto ad alcuna delle quattro operazioni fondamentali, come mostriamo nell'esempio seguente usando numeri decimali con tre cifre decimali:

$$600 + 600 = 1200 \text{ (troppo grande)}$$

$$003 - 005 = -2 \text{ (negativo)}$$

$$050 \times 050 = 2500 \text{ (troppo grande)}$$

$$007 / 002 = 3,5 \text{ (non è intero)}$$

Le violazioni della chiusura possono essere distinte in due classi disgiunte: le operazioni che danno un risultato più grande del massimo dell'insieme (*overflow*) o più piccolo del minimo dell'insieme (*underflow*), e le operazioni il cui risultato non appartiene all'insieme. Le prime tre violazioni dell'esempio sono del primo tipo, la quarta è del secondo.

I calcolatori dispongono di una memoria finita e quindi devono svolgere necessariamente conti aritmetici a precisione finita. Di conseguenza i risultati di alcuni calcoli saranno semplicemente sbagliati dal punto di vista della matematica classica. Un dispositivo di calcolo perfettamente funzionante che fornisca risultati errati può sembrare di primo acchito una stranezza, ma l'errore è una conseguenza logica della sua natura finita. Alcuni calcolatori dispongono di hardware speciale per il rilevamento di errori di overflow.

L'algebra dei numeri a precisione finita è diversa dall'algebra normale. Per esempio consideriamo l'espressione

$$a + (b - c) = (a + b) - c$$

(che è un'identità in base alla proprietà associativa di somma e sottrazione) e valutiamolo per  $a = 700$ ,  $b = 400$  e  $c = 300$ . Per calcolare il membro di sinistra cominciamo da  $(b - c)$ , che dà 100, e aggiungiamo questo valore ad  $a$ , per un totale di 800. Per calcolare il secondo membro, calcoliamo prima  $(a + b)$ , e ciò causa un overflow nell'aritmetica finita degli interi a tre cifre. Il risultato potrebbe dipendere dalla macchina utilizzata per il calcolo, ma non sarà mai 1100. La sottrazione di 300 da un qualunque numero diverso da 1100 non varrà mai 800. L'associatività non vale e l'ordine di esecuzione delle

operazioni è discriminante. Come ulteriore esempio, consideriamo la legge distributiva (del prodotto rispetto alla somma):

$$a \times (b - c) = a \times b - a \times c$$

e valutiamo entrambi i membri per  $a = 5$ ,  $b = 210$  e  $c = 195$ . Il primo membro vale  $5 \times 15$ , cioè 75, ma il secondo membro non dà 75 perché  $a \times b$  causa un overflow.

A giudicare da questi esempi, si potrebbe concludere che i calcolatori, pur essendo dei dispositivi per uso generale, sono poco portati per i calcoli aritmetici a causa della loro natura intrinsecamente finita. Ovvivamente questa conclusione è errata, ma serve a illustrare l'importanza di comprendere il funzionamento e le limitazioni dei calcolatori.

## A.2 Sistemi di numerazione in base fissa

Un numero decimale consiste in una sequenza di cifre decimali più, eventualmente, una virgola decimale. La Figura A.1 ne mostra la forma generale e la sua interpretazione posizionale. La scelta di 10 come base per l'elevamento a potenza dipende dal fatto che stiamo usando i decimali, ovvero i numeri in base 10. In informatica conviene spesso usare basi diverse da 10. Le basi più importanti sono 2, 8 e 16 e i sistemi di numerazione basati su di loro si chiamano rispettivamente binari, ottali ed esadecimali.

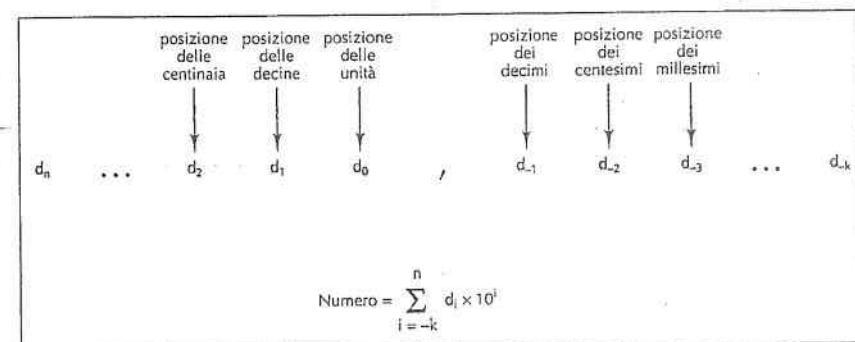


Figura A.1 Forma generale di un numero decimale.

Un sistema di numerazione in base  $k$  richiede  $k$  simboli diversi per rappresentare le cifre da 0 a  $k - 1$ . I numeri decimali si scrivono con le 10 cifre decimali

0 1 2 3 4 5 6 7 8 9

Invece i numeri binari non usano dieci cifre, ma si scrivono con le sole due cifre binarie

0 1

I numeri ottali si costruiscono a partire dalle otto cifre ottali

0 1 2 3 4 5 6 7

mentre i numeri esadecimales richiedono 16 cifre, cioè sei nuovi simboli. Per convenzione si usano le lettere maiuscole da A a F per le cifre che seguono 9. Dunque i numeri esadecimales si scrivono con le cifre

0 1 2 3 4 5 6 7 8 9 A B C D E F

Con l'espressione "bit" si intende una cifra che può assumere i valori 0 e 1. La Figura A.2 mostra il numero decimale 2001 espresso in binario, in ottale, in decimale e in forma esadecimale. Il numero 7B9 è ovviamente esadecimale, perché il simbolo B si trova solo nei numeri esadecimales. La stringa 111 invece può essere interpretata come un numero in uno qualsiasi dei quattro sistemi di numerazione presentati. Per evitare l'ambiguità, si usa indicare la base con un pedice 2, 8, 10 o 16, tutte le volte che non può essere intuita direttamente dal contesto.

Binario	1	1	1	1	1	0	1	0	0	0	0	1
	$1 \times 2^{10} + 1 \times 2^9 + 1 \times 2^8 + 1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 =$											
Ottale	1024	+ 512	+ 256	+ 128	+ 64	+ 0	+ 16	+ 0	+ 0	+ 0	+ 1	= 2001
	3	7	2	1								
	$3 \times 8^3 + 7 \times 8^2 + 2 \times 8^1 + 1 \times 8^0 =$											
	= 1536	+ 448	+ 16	+ 1								= 2001
Decimale	2	0	0	1								
	$2 \times 10^3 + 0 \times 10^2 + 0 \times 10^1 + 1 \times 10^0 =$											
	= 2000	+ 0	+ 0	+ 1								= 2001
Esadecimale	7	D	1									
	$7 \times 16^2 + 13 \times 16^1 + 1 \times 16^0 =$											
	= 1792	+ 208	+ 1									= 2001

Figura A.2 Il numero 2001 in binario, ottale, decimale ed esadecimale.

La Figura A.3 mostra un certo numero d'interi non negativi espressi nelle quattro basi. Forse un giorno, fra migliaia di anni, gli archeologi scopriranno questa tabella e la considereranno la Stele di Rosetta dei sistemi di numerazione usati a cavallo tra il tardo XX secolo e gli inizi del XXI secolo.

### A.3 Conversione tra basi

La conversione tra i sistemi ottale o esadecimale e binario è facile. Per convertire un numero binario in ottale basta suddividerlo in gruppi di 3 bit, avendo cura di raggruppare i 3 bit appena a sinistra (o a destra) della virgola e poi tutti gli altri bit. Ogni gruppo di 3 bit può essere convertito direttamente in una cifra ottale da 0 a 7, secondo la con-

versione indicata dalle prime righe della tabella nella Figura A.3. Potrebbe essere necessario aggiungere uno o più zeri in testa o in coda ai gruppi di bit che non hanno 3 cifre.

Decimale	Binario	Ottale	Esadecimale
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	3	3
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10
20	10100	24	14
30	11110	36	1E
40	101000	50	28
50	110010	62	32
60	111100	74	3C
70	1000110	106	46
80	1010000	120	50
90	1011010	132	5A
100	11001000	144	64
1000	1111101000	1750	3E8
2989	101110101101	5655	BAD

Figura A.3 Alcuni numeri decimali e i loro equivalenti binari, ottali, decimali ed esadecimali.

Anche la conversione da ottale a binario è banale. Ogni cifra ottale viene semplicemente rimpiazzata dall'equivalente numero binario a 3 bit. La conversione da esadecimale a binario è praticamente analoga a quella ottale-binario, con la differenza che le cifre esadecimales corrispondono a gruppi di 4 cifre binarie e non di 3. La Figura A.4 fornisce alcuni esempi di conversioni.

**Esempio 1**

Esadecimale

1	9	4	8	,	B	6
0001	1001	0100	1000.	1011	0110	00

1    4    5    1    0    ,    5    5    4

Binario

Ottale

**Esempio 2**

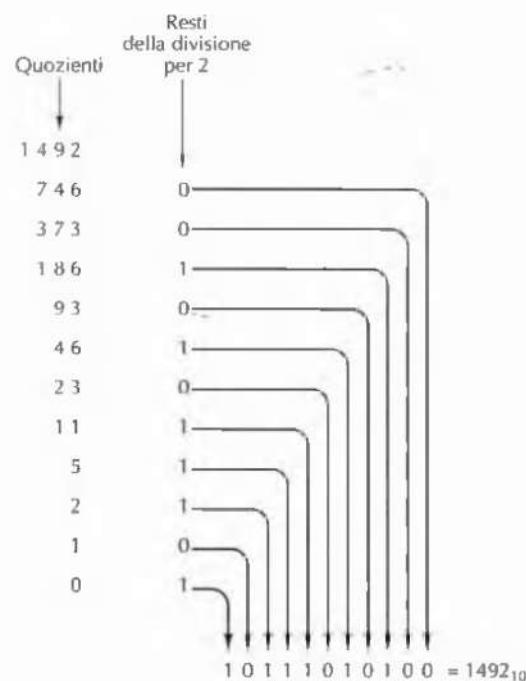
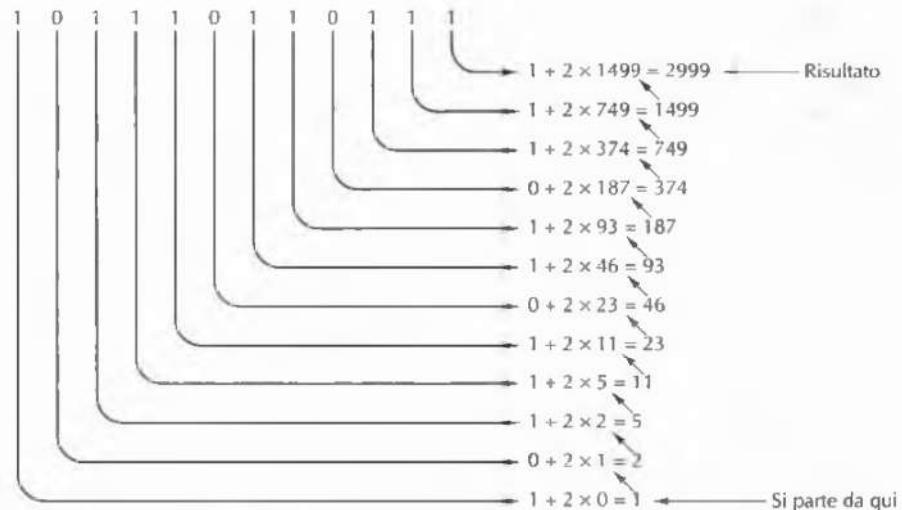
Esadecimale

7	B	A	3	,	B	C	4
0111	1011	0100	011,	1011	1100	0100	100

7    5    6    4    3    ,    5    7    0    4

Binario

Ottale

**Figura A.4** Esempi di conversione da ottale a binario e da esadecimale a binario.**Figura A.5** La conversione del numero decimale 1492 in binario mediante dimezzamenti successivi, partendo dall'alto e procedendo verso il basso. Per esempio, 93 diviso 2 fa 46 con resto 1, riportati nella riga successiva.**Figura A.6** La conversione del numero binario 101110110111 in decimale mediante raddoppiamenti successivi, a partire dal basso. Ogni riga si ottiene raddoppiando l'elemento della riga precedente e sommandogli il bit corrispondente. Per esempio, 749 è due volte 374 più il bit 1 che si trova in corrispondenza della riga di 749.

Il numero binario si ottiene direttamente scorrendo la colonna dei resti a partire dal basso. La Figura A.5 mostra un esempio di conversione da decimale a binario.

Anche gli interi binari possono essere convertiti in decimale con due metodi. Il primo consiste nel sommare le potenze di 2 che corrispondono alle posizioni degli 1 nel numero binario. Per esempio

$$10110 = 2^4 + 2^2 + 2^1 = 16 + 4 + 2 = 22$$

Nell'altro metodo, il numero binario si scrive verticalmente, un bit per riga, con il bit più significativo nell'ultima riga. Diciamo riga 1 l'ultima riga, riga 2 quella appena sopra e così via. Il numero decimale si costruisce in una colonna parallela a quella del numero binario. Si comincia con lo scrivere 1 nella riga 1. L'elemento della riga  $n$  è il doppio della riga  $n - 1$  più il bit della riga  $n$  (0 o 1). L'elemento della riga più in alto contiene il risultato della conversione. La Figura A.6 illustra un esempio di applicazione di questo metodo.

La conversione da decimale a ottale o esadecimale si può conseguire o passando per la conversione in binario, o mediante sottrazioni successive di potenze di 8 o di 16.

## A.4 Numeri binari negativi

Nei calcolatori digitali sono stati usati nel corso degli anni quattro diversi sistemi per la rappresentazione dei numeri negativi. Il primo si chiama **modulo e segno** e utilizza il bit più significativo come bit di segno (0 per il +, e 1 per il -) e i restanti come modulo (valore assoluto) del numero.

Il secondo sistema (che ormai è obsoleto) si chiama **complemento a uno** e anch'esso prevede un bit di segno dove 0 indica i numeri positivi, 1 i negativi. La negazione di un numero si ottiene scambiando tutti i suoi 1 con 0 e viceversa, compreso il bit di segno.

Il terzo sistema si chiama **complemento a due** e prevede un bit di segno, analogo al caso precedente. L'opposto di un numero si ottiene in due passi. Per prima cosa si rimpiazza ogni 1 con uno 0 e viceversa (proprio come nel complemento a uno) e poi si aggiunge 1 al risultato. Il risultato della somma binaria è lo stesso della somma decimale con la differenza che viene generato un resto se la somma è maggiore di 1, non di 9. Per esempio, la conversione di 6 in complemento a due si ottiene con i due passaggi:

00000110 (+6)  
11111001 (-6 in complemento a uno)  
11111010 (-6 in complemento a due).

L'eventuale resto in corrispondenza del bit più significativo viene ignorato.

Il quarto sistema si chiama **notazione in eccesso di  $2^{m-1}$**  (per numeri di  $m$  bit) e rappresenta il numero memorizzando la sua somma con  $2^{m-1}$ . Per esempio, nel caso di numeri di 8 bit ( $m = 8$ ) il sistema si dice essere in eccesso di 128 e memorizza un numero dopo avergli sommato 128. Così  $-3$  diventa  $-3 + 128 = 125$  e  $-3$  viene rappresentato dal numero binario di 8 bit che vale 125 (01111101). I numeri da  $-128$  a  $127$  corrispondono ai numeri da 0 a 255, tutti esprimibili come interi positivi di 8 bit. È abbastanza interessante notare che questo sistema è identico al complemento a due con il bit di segno invertito. La Figura A.7 fornisce alcuni esempi di numeri negativi espressi con i quattro sistemi.

Il modulo con segno e il complemento a uno hanno entrambi due distinte rappresentazioni di zero: uno zero positivo e uno zero negativo. Questa proprietà non è molto gradita. Il sistema in complemento a due non è affatto da questo problema, perché il complemento a due di 0 è ancora 0. Tuttavia anche il sistema in complemento a due ha una sua anomalia: la configurazione formata da un 1 seguito da tutti 0 è il complemento di se stessa. L'effetto di questa proprietà è che l'intervallo dei numeri positivi e quello dei numeri negativi è asimmetrico: esiste un numero negativo che non ha una controparte tra i numeri positivi.

La motivazione che sta dietro a questi problemi è facile da individuare: desideriamo un sistema di codifica che esibisca le due proprietà seguenti:

1. una sola rappresentazione dello zero
2. ugualc numero d'interi positivi e interi negativi rappresentati.

N decimale	N binario	-N modulo con segno	-N compl. a 1	-N compl. a 2	-N in eccesso di 128
1	00000001	10000001	11111110	11111111	01111111
2	00000010	10000010	11111101	11111110	01111110
3	00000011	10000011	11111100	11111101	01111101
4	00000100	10000100	11111011	11111100	01111100
5	00000101	10000101	11111010	11111011	01111011
6	00000110	10000110	11111001	11111010	01111010
7	00000111	10000111	11111000	11111001	01111001
8	00001000	10001000	11110111	11111000	01111000
9	00001001	10001001	11110110	11110111	01110111
10	00001010	10001010	11110101	11110110	01110110
20	00010100	10010100	11101011	11101100	01101100
30	00011110	10011110	11100001	11100010	01100010
40	00101000	10101000	11010111	11011000	01011000
50	00110010	10110010	11001101	11001110	01001110
60	00111100	10111100	11000011	11000100	01000100
70	01000110	11000110	10111001	10111010	00111010
80	01010000	11010000	10101111	10110000	00110000
90	01011010	11011010	10100101	10100110	00100110
100	01100100	11100100	10011011	10011100	00011100
127	01111111	11111111	10000000	10000001	00000001
128	Inesistente	Inesistente	Inesistente	10000000	00000000

Figura A.7 Numeri negativi di 8 bit nei quattro sistemi di rappresentazione.

Il fatto è che un insieme di numeri che abbia lo stesso numero di elementi positivi e negativi e una sola rappresentazione di 0 ha necessariamente un numero dispari di elementi, mentre con  $m$  bit si può rappresentare un numero pari di stringhe di bit. Perciò, qualunque rappresentazione conterrà sempre una configurazione in più (o in meno) di quanto desiderato. La configurazione eccedente può essere utilizzata per rappresentare  $-0$ , per denotare un numero negativo grande (in modulo) o per qualcos'altro; comunque sia resterà sempre una seccatura.

Infine va ricordato che, data una stringa di  $n$  bit in complemento a 2, diciamo

$$b_{n-1} \dots b_1 b_0,$$

il numero da essa rappresentato è ottenuto mediante la formula

$$-b_{n-1} \times 2^{n-1} + \sum_{i=0}^{n-2} b_i \times a^i.$$

Addendo	$0 +$	$0 +$	$1 +$	$1 +$
Addendo	$0 =$	$1 =$	$0 =$	$1 =$
Somma	0	1	1	0
Riporto	0	0	0	1

Figura A.8 Tabellina della somma binaria.

## A.5 Aritmetica binaria

La Figura A.8 riporta la tabellina della somma per i numeri binari.

La somma di due addendi binari inizia dal bit meno significativo e procede sommando i bit che si trovano nelle posizioni corrispondenti. Se c'è un riporto, lo si somma nella colonna successiva a sinistra, proprio come nell'aritmetica decimale. Nell'aritmetica in complemento a uno, l'eventuale riporto generato dalla somma dei bit più significativi viene sommato al bit meno significativo. Questo procedimento si chiama *end-around carry* (cioè "riporto circolare"). Nell'aritmetica in complemento a due, l'eventuale riporto generato dalla somma dei bit più significativi viene semplicemente scartato. La Figura A.9 mostra due esempi di operazioni binarie.

Se gli addendi hanno segno opposto non si può verificare overflow. Se invece hanno lo stesso segno e il risultato ha segno opposto si è verificato un overflow e il risultato della somma è sbagliato. In entrambi i sistemi in complemento, si ha overflow se e soltanto se il resto in corrispondenza del bit di segno differisce dal riporto generato oltre il bit di segno. Molti calcolatori conservano questo bit in un apposito bit di overflow, ma il riporto del bit di segno non è desumibile dal risultato.

Decimale	In complemento a 1	In complemento a 2
$10 +$	$00001010 +$	$00001010 +$
$(-3) =$	$11111100 =$	$11111101 =$
$+7$	$1\ 00000110$	$1\ 00000111$
	riporto 1	scartato
		$00000111$

Figura A.9 Somma in complemento a uno e in complemento a due.

## Problemi

- Si convertano in binario i numeri: 1984, 4000, 8192.
- Qual è il numero decimale, ottale, esadecimale, rappresentato dalla stringa binaria 1001101001?
- Quali delle seguenti stringhe

ADA, BABBO, BARBA, CACCIA, DADA, DIDA, DECADE, EFFE, rappresentano un numero esadecimale?

- Si esprima il numero decimale 100 in tutte le basi dalla 2 alla 9.
- Quanti numeri diversi si possono rappresentare con  $k$  cifre in base  $b$ ?
- Molte persone riescono a contare solo fino a 10 solo usando le dita, ma gli informatici sanno fare di meglio. Se si guarda a ogni dito come a una cifra binaria, dove un dito teso indica 1 e un dito piegato verso il palmo indica 0, fino a che numero è possibile contare usando entrambe le mani? Con le mani e i piedi? Si usino ora le mani e i piedi per rappresentare i numeri in complemento a due, con l'alluce del piede sinistro impiegato come bit di segno. Qual è l'intervallo dei numeri rappresentabili? (\*)
- Si svolgano le operazioni seguenti su numeri di 8 bit in complemento a due.

$$\begin{array}{r} 00101101 + \\ 01101111 = \\ \hline \end{array} \quad \begin{array}{r} 11111111 + \\ 11111111 = \\ \hline \end{array} \quad \begin{array}{r} 00000000 + \\ 11110111 = \\ \hline \end{array} \quad \begin{array}{r} 11110111 + \\ 11110111 = \\ \hline \end{array}$$

.....

- Si svolga l'esercizio precedente nell'aritmetica in complemento a uno.
- Si considerino le seguenti somme di numeri binari di 3 bit in complemento a due. Per ogni somma, si stabilisca se:

- il bit di segno del risultato è 1
- i tre bit meno significativi valgono 0
- si verifica un overflow.

$$\begin{array}{r} 000 + \\ 001 = \\ \hline \end{array} \quad \begin{array}{r} 000 + \\ 111 = \\ \hline \end{array} \quad \begin{array}{r} 111 + \\ 110 = \\ \hline \end{array} \quad \begin{array}{r} 100 + \\ 111 = \\ \hline \end{array} \quad \begin{array}{r} 100 + \\ 100 = \\ \hline \end{array}$$

.....

- I numeri decimali con segno lunghi  $n$  cifre possono essere rappresentati senza segno con  $n+1$  cifre. I numeri positivi hanno 0 come cifra più significativa, mentre i numeri negativi si ottengono a partire dai positivi sottraendo ciascuna loro cifra dalla cifra 9. Dunque la negazione di 014725 è 985274. Questa rappresentazione si chiama complemento a nove ed è analoga al complemento a uno dei numeri binari. Si esprimano i numeri seguenti come numeri di tre cifre in complemento a nove: 6, -2, 100, -14, -1, 0.

(\*) Si declina ogni responsabilità sulle conseguenze di natura traumatico-ortopedica cui possono incorrere i lettori che svolgono meticolosamente questo esercizio (N.d.R.).

11. Si determini la regola della somma di numeri in complemento a nove e la si utilizzi sulle seguenti coppie di numeri:

$$\begin{array}{r} 0001 + \\ 9999 = \end{array}$$

$$9997 + 9996 = 19993$$

12. Il complemento a dieci è analogo al complemento a due. Un numero negativo in complemento a dieci si ottiene sommando 1 al numero corrispondente in complemento a nove, ignorando l'eventuale resto. Qual è la regola della somma in complemento a dieci?
  13. Si rediga la tabellina della moltiplicazione dei numeri in base 3.
  14. Si moltipichi in binario 0111 e 0011.
  15. Si scriva un programma che accetti in ingresso un numero decimale con segno, sotto forma di stringa di caratteri ASCII, e stampi a schermo la sua rappresentazione in binario in complemento a due, in ottale e in esadecimale.
  16. Si scriva un programma che prenda in ingresso due stringhe di 32 bit, sotto forma di caratteri ASCII. Il programma deve stampare la stringa di 32 caratteri ASCII corrispondente alla somma dei numeri, in complemento a 2, rappresentati dalle stringhe in ingresso.

## Appendice B

### Numeri in virgola mobile

In molti calcoli l'intervallo dei numeri utilizzati è molto esteso. Per esempio, un calcolo astronomico potrebbe riguardare la massa dell'elettrone,  $9 \times 10^{-28}$  grammi, e la massa del sole,  $2 \times 10^{33}$  grammi, con una differenza tra i valori superiore a  $10^{60}$ . Questi due valori potrebbero essere rappresentati nella seguente forma

e tutti i calcoli potrebbero essere eseguiti mantenendo 34 cifre alla sinistra della virgola decimal e 28 posizioni alla sua destra. Così facendo i risultati sarebbero composti da 62 cifre significative. Su un calcolatore si potrebbe utilizzare l'aritmetica multiprecisione per fornire un numero sufficiente di cifre significative. Tuttavia la massa del sole non è conosciuta in modo accurato neanche fino alla quinta cifra, e tanto meno lo può essere fino alla sessantaduesima. In realtà poche misurazioni, di qualsiasi tipo esse siano, possono essere effettuate con (o richiedono una) precisione di 62 cifre significative. In teoria si potrebbero utilizzare tutte e 62 le cifre per i risultati intermedi per poi scartarne 50 o 60 prima di stampare i risultati finali, ma in pratica ciò non sarebbe altro che uno spreco di memoria e di tempo di CPU.

Quello di cui abbiamo bisogno è un sistema di rappresentazione dei numeri, nel quale l'intervallo di valori esprimibili sia indipendente dal numero di cifre significative. In questa appendice presenteremo un simile sistema, che si basa sulla notazione scientifica usata comunemente in fisica, chimica e ingegneria.

## B.1 Principi dell'aritmetica in virgola mobile

Un modo per disaccoppiare l'intervallo dalla precisione consiste nell'esprimere i numeri nella notazione scientifica a noi familiare

$$n = f \times 10^e$$

dove  $f$  è chiamata **frazione**, o **mantissa**, ed  $e$  è un intero positivo o negativo chiamato **esponente**. La versione per il calcolatore di questa notazione è chiamata rappresentazione in **virgola mobile**. Alcuni esempi di numeri espressi in questa forma sono

$$\begin{aligned} 3,14 &= 0,314 \times 10^1 = 3,14 \times 10^0 \\ 0,000001 &= 0,1 \times 10^{-5} = 1,0 \times 10^{-6} \\ 1941 &= 0,1941 \times 10^4 = 1,941 \times 10^3 \end{aligned}$$

L'intervallo è determinato dal numero di cifre che compongono l'esponente, mentre la precisione è determinata dal numero di cifre della frazione. Dato che un numero può essere rappresentato in più modi, in genere si sceglie un'unica forma come standard. Per analizzare le proprietà di cui gode questo metodo di rappresentazione dei numeri, consideriamo una rappresentazione,  $R$ , in cui la frazione vale zero oppure un valore con segno a tre cifre compreso nell'intervallo  $0,1 \leq |f| < 1$ , mentre l'esponente è un valore con segno a due cifre. Questi numeri variano, in modulo, da  $0,100 \times 10^{-99}$  a  $0,999 \times 10^{99}$  e, nonostante una variazione di quasi 199 ordini di grandezza, per essere memorizzati essi richiedono solamente cinque cifre e due segni +.

I numeri in virgola mobile sono utilizzabili per modellare i numeri reali, anche se in modo impreciso. La Figura B.1 mostra uno schema, molto semplificato, della retta reale. Questa linea è divisa in sette regioni:

1. grandi numeri negativi minori di  $-0,999 \times 10^{99}$
2. numeri negativi compresi tra  $-0,999 \times 10^{99}$  e  $-0,100 \times 10^{-99}$
3. piccoli numeri negativi con modulo minore di  $0,100 \times 10^{-99}$
4. zero
5. piccoli numeri positivi con modulo minore di  $0,100 \times 10^{-99}$
6. numeri positivi compresi tra  $0,100 \times 10^{-99}$  e  $0,999 \times 10^{99}$
7. grandi numeri positivi maggiori di  $0,999 \times 10^{99}$ .

Un'importante differenza tra i numeri reali e l'insieme di numeri rappresentabili usando tre cifre per la frazione e due per l'esponente è che in tal modo non si possono esprimere i numeri appartenenti alle regioni 1, 3, 5 e 7. Se il risultato di un'operazione aritmetica è un nu-

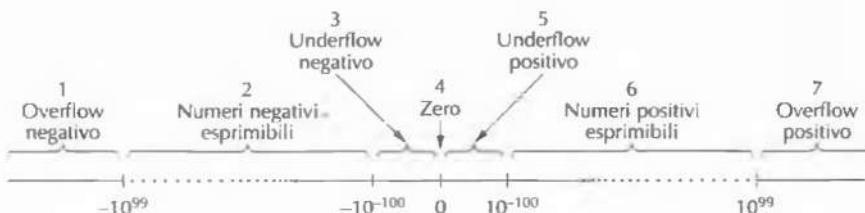


Figura B.1 La rappresentazione R ripartisce la retta reale in sette regioni.

mero contenuto nelle regioni 1 oppure 7, per esempio  $10^{60} \times 10^{60} = 10^{120}$ , si verifica un **errore di overflow** e la risposta non sarà corretta. Il motivo è legato alla natura finita della rappresentazione dei numeri ed è inevitabile. È altrettanto impossibile esprimere un risultato che cade nella regione 3 oppure nella 5, una situazione chiamata **errore di underflow**. Questo errore è meno serio di quello di overflow, dato che spesso 0 è un'approssimazione soddisfacente per i numeri delle regioni 3 e 5. Infatti un saldo bancario di  $10^{-102}$  euro non è molto migliore di uno di 0 euro.

Un'altra importante differenza tra i numeri in virgola mobile e i numeri reali è la loro densità. Tra due qualsiasi numeri reali,  $x$  e  $y$ , esiste un altro numero reale, indipendentemente da quanto siano vicini  $x$  e  $y$ . Questa proprietà deriva dal fatto che per ogni coppia di numeri reali distinti,  $x$  e  $y$ ,  $z = (x + y)/2$  è un numero reale compreso fra  $x$  e  $y$ . I numeri reali formano un continuo.

Al contrario i numeri in virgola mobile sono un sistema discreto. Nel sistema a cinque cifre e due segni usato precedentemente è possibile esprimere esattamente 179.100 numeri positivi, 179.100 numeri negativi e il valore 0 (che può essere espresso in molti modi), per un totale di 358.201 numeri. Usando questa notazione è possibile quindi esprimere solamente 358.201 numeri degli infiniti numeri reali compresi tra  $-10^{100}$  e  $+0,999 \times 10^{99}$ . Essi sono rappresentati dai punti della Figura B.1. È molto probabile che il risultato di un calcolo, pur facendo parte delle regioni 2 o 6, sia uno degli altri numeri, quelli non rappresentabili. Per esempio  $+0,100 \times 10^3$  diviso 3 non può essere espresso *esattamente* nel nostro sistema di rappresentazione. Se il risultato di un calcolo non può essere espresso nella rappresentazione utilizzata, la soluzione più ovvia consiste nell'utilizzare il più vicino numero rappresentabile. Questo processo è chiamato **arrotondamento**.

All'interno delle regioni 2 e 6 lo spazio tra numeri adiacenti non è costante. La distanza tra  $+0,998 \times 10^{99}$  e  $+0,999 \times 10^{99}$  è enormemente più grande della distanza tra  $+0,998 \times 10^0$  e  $+0,999 \times 10^1$ . Tuttavia, se si esprime la distanza tra un numero e il suo successore come una percentuale di quel valore non vi è una sistematica variazione lungo le regioni 2 e 6. In altre parole l'**errore relativo** introdotto dall'arrotondamento è approssimativamente lo stesso sia per numeri piccoli sia per quelli grandi.

Cifre nella mantissa	Cifre nell'esponente	Limite inferiore	Limite superiore
3	1	$10^{-12}$	$10^9$
3	2	$10^{-102}$	$10^{99}$
3	3	$10^{-1002}$	$10^{999}$
3	4	$10^{-10002}$	$10^{9999}$
4	1	$10^{-13}$	$10^9$
4	2	$10^{-103}$	$10^{99}$
4	3	$10^{-1003}$	$10^{999}$
4	4	$10^{-10003}$	$10^{9999}$
5	1	$10^{-14}$	$10^9$
5	2	$10^{-104}$	$10^{99}$
5	3	$10^{-1004}$	$10^{999}$
5	4	$10^{-10004}$	$10^{9999}$
10	3	$10^{-109}$	$10^{99}$
20	3	$10^{-1019}$	$10^{999}$

Figura B.2 Limiti inferiore e superiore approssimati dei numeri decimali (non normalizzati) esprimibili in virgola mobile.

Anche se queste considerazioni sono basate su una rappresentazione con tre cifre per la mantissa e due cifre per l'esponente, le conclusioni tratte valgono anche per altri sistemi di rappresentazione. Modificando il numero di cifre della frazione o dell'esponente si spostano semplicemente i limiti delle regioni 2 e 6 e si modifica il numero di punti esprimibili al loro interno. Aumentando il numero di cifre della frazione si aumenta la densità dei punti e quindi si migliora il grado di accuratezza dell'approssimazione. Aumentando le cifre dell'esponente si aumenta la dimensione delle regioni 2 e 6 riducendo le regioni 1, 3, 5 e 7. La Figura B.2 mostra, in modo approssimato, i limiti della regione 6 per numeri decimali in virgola mobile che utilizzano dimensioni diverse per la frazione e l'esponente.

Nei calcolatori si utilizza una variante di questa rappresentazione. Per ragioni di efficienza l'elevamento a potenza viene fatto utilizzando come base 2, 4, 8 o 16, e non 10: la frazione consiste quindi in una stringa di cifre binarie, in base 4, ottali o esadecimali. Se la cifra più a sinistra è zero è possibile traslare di una posizione a sinistra tutte le cifre e diminuire di 1 l'esponente senza variare il valore del numero (a parte l'underflow). Una frazione in cui la cifra più a sinistra è diversa da zero è detta **normalizzata**.

Generalmente i numeri normalizzati sono preferibili a quelli non normalizzati, dato che dei primi esiste un'unica forma, mentre dei secondi ce ne sono molteplici. La Figura B.3 mostra un numero in virgola mobile normalizzato con due basi diverse per l'elevamento a potenza. Negli esempi è stata utilizzata una rappresentazione in cui la frazione usa 16 bit (compreso il bit del segno) e l'esponente usa 7 bit in notazione in eccesso 64.

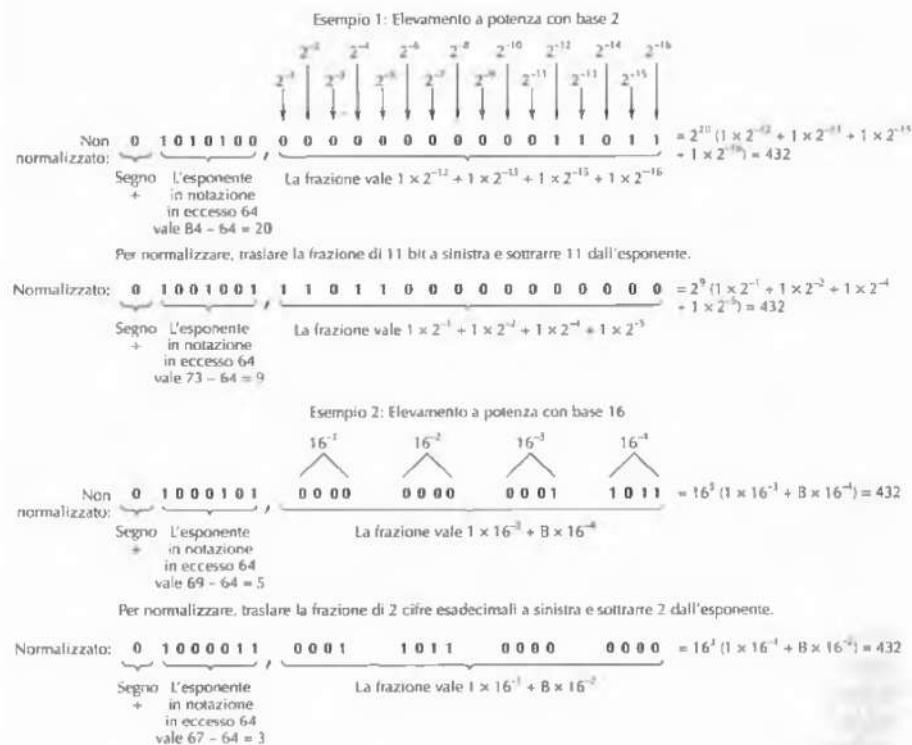
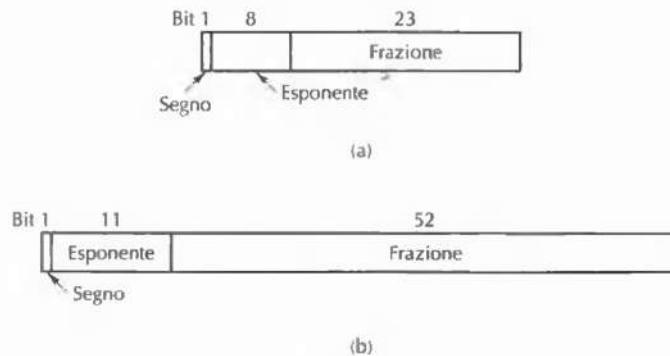


Figura B.3 Esempi di numeri in virgola mobile normalizzati.

## B.2 Standard in virgola mobile IEEE 754

Fino al 1980 ogni produttore di calcolatori aveva il proprio formato di aritmetica in virgola mobile che, non c'è bisogno di dirlo, erano tutti diversi l'uno dall'altro. Ancor peggio, alcune di queste rappresentazioni non eseguivano correttamente i calcoli aritmetici; esistono infatti alcune finezze dell'aritmetica in virgola mobile che non appaiono così ovvie a tutti i progettisti hardware.

Alla fine degli anni '70, per porre rimedio alla situazione, venne costituito un comitato per standardizzare l'aritmetica in virgola mobile. L'obiettivo non era solo quello di permettere a calcolatori diversi di scambiarsi dati in virgola mobile, ma anche quello di fornire ai progettisti dell'hardware un modello di cui si conosceva la correttezza. Il risultato del lavoro fu lo standard IEEE 754 (IEEE, 1985). La maggior parte dei calcolatori attuali (compresi i modelli Intel, SPARC e JVM studiati in questo testo) ha istruzioni in virgola mobile



**Figura B.4** Formato IEEE in virgola mobile. (a) Precisione singola. (b) Precisione doppia.

conformi a questo standard. Diversamente da molti altri standard che tendono a essere soltanto dei compromessi annacquati che non accontentano nessuno, questo standard non è male; il motivo principale è che fu il prodotto del lavoro di una sola persona: il professor William Kahan, un matematico di Berkely.

Lo standard definisce tre formati: precisione singola (32 bit), precisione doppia (64 bit) e precisione estesa (80 bit). Quest'ultimo è stato pensato per ridurre gli errori di arrotondamento ed è usato principalmente nelle unità aritmetiche in virgola mobile; per questo motivo non lo tratteremo ulteriormente. I primi due (mostrati nella Figura B.4) usano la base 2 per le frazioni e la notazione in eccesso per gli esponenti.

Entrambi i formati iniziano con un bit che rappresenta il segno dell'intero numero, 0 se è positivo e 1 se è negativo. Di seguito vi è l'esponente, che utilizza la notazione in eccesso 127 per la precisione singola e la notazione in eccesso 1023 per la precisione doppia. Gli esponenti minimo (0) e massimo (255 e 2047) non sono utilizzati per i numeri normalizzati; essi hanno un utilizzo speciale che descrivremo in seguito. Infine troviamo le frazioni, rispettivamente di 23 e 52 bit.

Una frazione normalizzata inizia con una virgola, seguita da un bit 1 e poi dal resto della frazione. Seguendo una pratica iniziata con il PDP-11 gli autori dello standard hanno deciso che il bit 1 che dovrebbe precedere la virgola non doveva essere memorizzato, dato che se ne poteva implicitamente assumere la presenza. Lo standard definisce quindi la frazione in modo leggermente diverso dal solito; consiste in un bit 1 sottointeso, in una virgola binaria sottointesa e poi in 23 (o 52) bit arbitrari. Se tutti e 23 (o 52) i bit della frazione valgono 0, il valore numerico della frazione è 1,0. Se tutti valgono 1, il valore numerico della frazione è leggermente inferiore a 2,0. Per evitare confusione con la forma convenzionale della frazione, la combinazione del valore 1 sottointeso, della virgola binaria sottointesa e dei 23, o 52, bit espliciti è chiamata **significando**, invece di frazione o mantissa. Tutti i numeri normalizzati hanno un significando,  $s$ , compreso nell'intervallo [1,2].

Le caratteristiche numeriche dello standard sono elencate nella Figura B.5. Come esempio consideriamo i numeri 0,5, 1 e 1,5 nel formato normalizzato in precisione singola. Le loro rappresentazioni in esadecimale sono rispettivamente 3F000000, 3F800000 e 3FC00000.

Entrambi i formati iniziano con un bit che rappresenta il segno dell'intero numero, 0 se è positivo e 1 se è negativo. Di seguito vi è l'esponente, che utilizza la notazione in eccesso 127 per la precisione singola e la notazione in eccesso 1023 per la precisione doppia. Gli esponenti minimo (0) e massimo (255 e 2047) non sono utilizzati per i numeri normalizzati; essi hanno un utilizzo speciale che descriveremo in seguito. Infine troviamo le frazioni, rispettivamente di 23 e 52 bit.

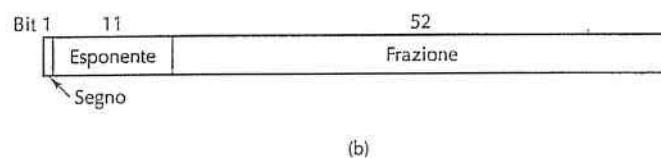
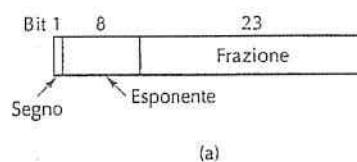


Figura B.4 Formato IEEE in virgola mobile. (a) Precisione singola. (b) Precisione doppia.

Una frazione normalizzata inizia con una virgola, seguita da un bit 1 e poi dal resto della frazione. Seguendo una pratica iniziata con il PDP-11 gli autori dello standard hanno deciso che il bit 1 che dovrebbe precedere la virgola non doveva essere memorizzato, dato che se ne poteva implicitamente assumere la presenza. Lo standard definisce quindi la frazione in modo leggermente diverso dal solito; consiste in un bit 1 sottointeso, in una virgola binaria sottointesa e poi in 23 (o 52) bit arbitrari. Se tutti e 23 (o 52) i bit della frazione valgono 0, il valore numerico della frazione è 1,0. Se tutti valgono 1, il valore numerico della frazione è leggermente inferiore a 2,0. Per evitare confusione con la forma convenzionale della frazione, la combinazione del valore 1 sottointeso, della virgola binaria sottointesa e dei 23, o 52, bit espliciti è chiamata significando, invece di frazione o mantissa. Tutti i numeri normalizzati hanno un significando,  $s$ , compreso nell'intervallo [1,2].

Le caratteristiche numeriche dello standard sono elencate nella Figura B.5. Come esempio consideriamo i numeri 0,5, 1 e 1,5 nel formato normalizzato in precisione singola. Le loro rappresentazioni in esadecimale sono rispettivamente 3F000000, 3F800000 e 3FC00000.

Uno dei tipici problemi dell'aritmetica in virgola mobile è il modo di gestire l'underflow, l'overflow e i numeri non inizializzati. Lo standard IEEE tratta questi problemi in modo esplicito, traendo parte del proprio approccio da quello utilizzato nel calcolatore CDC 6600. Oltre ai numeri normalizzati lo standard definisce quattro altri tipi numerici, descritti in seguito e mostrati nella Figura B.6.

Caratteristica	Precisione singola	Precisione doppia
Bit nel segno	1	1
Bit nell'esponente	8	11
Bit nella frazione	23	52
Bit totali	32	64
Notazione dell'esponente	in eccesso 127	in eccesso 1023
Intervallo dell'esponente	da -126 a +127	da -1022 a +1023
Numero normalizzato più piccolo	$2^{-126}$	$2^{-1022}$
Numero normalizzato più grande	circa $2^{128}$	circa $2^{1024}$
Intervallo decimale	circa da $10^{-38}$ a $10^{38}$	circa da $10^{-308}$ a $10^{308}$
Numero denormalizzato più grande	circa $10^{-45}$	circa $10^{-324}$

Figura B.5 Caratteristiche dell'aritmetica IEEE in virgola mobile.

Normalizzato	$\pm$ $0 < \text{Esp} < \text{Max}$	qualsiasi stringa di bit
Denormalizzato	$\pm$ 0	qualsiasi stringa di bit diversa da zero
Zero	$\pm$ 0	0
Infinito	$\pm$ 1 1 1...1	0
Not A Number	$\pm$ 1 1 1...1	qualsiasi stringa di bit diversa da zero

Bit di segno

Figura B.6 Tipi di numeri IEEE.

Quando il risultato di un calcolo ha un modulo minore del più piccolo numero in virgola mobile normalizzato sorge un problema, dato che questo sistema non può rappresentarlo.

Prima della definizione dello standard IEEE le scelte più comuni erano: impostare il risultato a zero e continuare oppure generare un'eccezione di underflow. Dato che nessuna di queste soluzioni era veramente soddisfacente, IEEE ha inventato i numeri denormalizzati. Questi numeri hanno come esponente 0 e come frazione i successivi 23 (o 52) bit. Il bit 1 implicito alla sinistra della virgola binaria diventa ora 0. È possibile distinguere i numeri normalizzati da quelli denormalizzati, dato che gli ultimi non possono avere un valore 0 come esponente.

Il più piccolo numero normalizzato in precisione singola ha 1 come esponente e 0 come frazione, e rappresenta il numero  $1,0 \times 2^{-126}$ . Il più grande numero denormalizzato

ha 0 come esponente e tutti i bit della frazione e impostati a 1, e rappresenta circa  $0,9999999 \times 2^{-126}$ , che è praticamente uguale al valore precedente. Occorre tuttavia notare che questo numero ha solo 23 bit significativi, contro i 24 dei numeri normalizzati.

Al diminuire del risultato dei calcoli l'esponente rimane fisso a 0, mentre si azzerano progressivamente i primi bit della frazione, riducendo in questo modo sia il valore rappresentato sia il numero di bit significativi della frazione. Il più piccolo numero denormalizzato diverso da zero consiste in un valore 1 nel bit più a destra, con tutti gli altri bit pari a 0. Dato che l'esponente rappresenta  $2^{-126}$  e la frazione rappresenta  $2^{-23}$ , il valore è  $2^{-149}$ . Questo scherzo rende meno grave il problema dell'underflow, dato che quando il risultato non può essere espresso come un risultato normalizzato non si salta improvvisamente al valore 0, ma il numero di cifre significative diminuisce progressivamente.

In questo schema lo zero ha due rappresentazioni, una positiva e una negativa, determinate dal bit del segno. Entrambe hanno 0 come esponente e come frazione. Anche in questo caso il bit a sinistra della virgola binaria è implicitamente 0 invece di 1.

L'overflow non può essere gestito in modo graduale come l'underflow, dato che non ci sono altre combinazioni di bit. Esiste però una rappresentazione per l'infinito che consiste in un esponente in cui tutti i bit valgono 1 (non permesso per i numeri normalizzati) e di una frazione che vale 0. Questo numero segue le regole matematiche dell'infinito. Per esempio la somma tra infinito e qualsiasi altro valore dà come risultato infinito e qualsiasi numero finito diviso infinito dà come risultato zero. Analogamente ogni numero finito diviso zero dà come risultato infinito.

E se si divide infinito per infinito? Il risultato non è definito. Per gestire questo caso è previsto un ulteriore formato, chiamato NaN (Not a Number). Anch'esso può essere usato come operando con risultati prevedibili.

### PROBLEMI

- Si convertano i seguenti numeri nel formato IEEE in precisione singola. Si scriva il risultato in esadecimale.
  - 9
  - $5/32$
  - $-5/32$
  - $6,125$
- Si convertano da esadecimali a decimali i seguenti numeri rappresentati nel formato IEEE in virgola mobile in precisione singola:
  - $42E48000H$
  - $3F880000H$
  - $00800000H$
  - $C7F00000H$
- Sul calcolatore 370 il formato dei numeri in virgola mobile in precisione singola è composto da un esponente di 7 bit in notazione in eccesso 64 e da una frazione contenente 24 bit più un bit per il segno; la virgola si trova all'estremità sinistra della frazione. La base dell'elevamento a potenza è 16. L'ordine dei campi è: bit del segno, esponente, frazione. Si esprima, in esadecimale, il numero  $7/64$  come un numero normalizzato in questo sistema.

- I seguenti numeri binari in virgola mobile consistono in un bit del segno, un esponente di una base 2 espresso in notazione in eccesso 64 e una frazione a 16 bit. Li si normalizzi.
  - 0 1000000 000101010000001
  - 0 0111111 000001111111111
  - 0 1000011 1000000000000000
- Per sommare due numeri in virgola mobile occorre modificare gli esponenti (traslando i bit della frazione) affinché abbiano lo stesso valore. In seguito è possibile sommare le frazioni e, se necessario, normalizzare il risultato. Si sommino i numeri IEEE in precisione singola  $3EE00000H$  e  $3D800000H$  e si esprima in esadecimale il risultato normalizzato.
- La Società di Calcolatori BraccioCorto ha deciso di produrre una macchina con numeri in virgola mobile a 16 bit. Il formato in virgola mobile del Modello 0,001 ha un bit per il segno, 7 bit per l'esponente in notazione in eccesso 64 e 8 bit per la frazione. Il formato del Modello 0,002 ha un bit per il segno, 5 bit per l'esponente in notazione in eccesso 16 e 10 bit per la frazione. Entrambi usano la base 2 per l'elevamento a potenza. Quali sono i numeri più piccoli e più grandi rappresentabili sui due modelli? Quante cifre decimali di precisione ha, all'incirca, ciascuno di loro? Comprereste uno di questi modelli?
- Esiste una situazione nella quale un'operazione su due numeri in virgola mobile possa causare una riduzione drastica del numero di bit significativi nel risultato. Qual è?
- Alcuni coprocessori in virgola mobile hanno un'istruzione predefinita per la radice quadrata. Un algoritmo possibile è di tipo iterativo (per esempio, Newton-Raphson). Gli algoritmi ite radici hanno bisogno di un'approssimazione iniziale del risultato che poi migliorano costantemente. Come si può ottenere una veloce approssimazione della radice quadrata di un numero in virgola mobile?
- Si scriva una procedura per sommare due numeri IEEE in virgola mobile in precisione singola. Ciascun numero è rappresentato da un array di 32 elementi booleani.
- Si scriva una procedura per sommare due numeri in virgola mobile in precisione singola che usano la base 16 per l'esponente e la base 2 per la frazione, ma non hanno un bit 1 sottointeso alla sinistra della virgola. Un numero normalizzato ha, nei quattro bit più a sinistra della frazione, 0001, 0010, ..., 1111, ma non 0000. Un numero viene normalizzato traslando a sinistra di 4 bit la frazione e sottraendo 1 all'esponente.

## Appendice C

# Programmazione in linguaggio assemblativo

Evert Wattel  
Vrije Universiteit  
Amsterdam, Olanda

Ogni calcolatore ha un livello **ISA** (*Instruction Set Architecture*, "architettura dell'insieme d'istruzioni") che riguarda registri, istruzioni e altre caratteristiche visibili ai programmatore di linguaggi a basso livello. Spesso ci si riferisce al livello ISA con il termine di **linguaggio macchina**, benché non sia del tutto corretto. Un programma a questo livello di astrazione è una lunga lista di stringhe binarie, una per istruzione, che specificano le istruzioni da eseguire e i loro operandi. Programmare in binario è molto astruso, perciò tutti i calcolatori dispongono di un **linguaggio assemblativo** (*assembly language*), cioè di una rappresentazione simbolica dell'architettura dell'insieme d'istruzioni che usa nomi quali ADD, SUB e MUL al posto delle stringhe binarie. Questa appendice tratta la programmazione in linguaggio assemblativo dell'Intel 8088 (il processore utilizzato nei primi PC di IBM e che costituì la base per lo sviluppo dei Pentium moderni) e l'uso di alcuni strumenti disponibili nel materiale di supporto allegato al testo.

Il nostro scopo non è formare nuovi *programmatore assembler* (questa è la locuzione gergale), ma assistere il lettore nell'esplorazione dell'architettura dei calcolatori mettendolo in grado di "toccare con mano" la materia. A tal fine, abbiamo scelto come esempio della trattazione una macchina molto semplice. Anche se è oramai impossibile procurarsi questi processori, tutti i Pentium possono eseguire i programmi scritti per l'8088 e quindi le lezioni che impartiremo valgono ancora per le macchine moderne. Inoltre, molte delle istruzioni principali del Pentium sono uguali a quelle dell'8088, con la sola differenza che usano registri a 32 bit invece che a 16. Dunque questa appendice può essere considerata una graduale introduzione alla programmazione nel linguaggio assemblativo del Pentium.

La programmazione di una macchina in linguaggio assemblativo prevede che il programmatore abbia una conoscenza approfondita dell'architettura dell'insieme d'istruzioni della macchina. Di conseguenza, i primi quattro paragrafi sono dedicati all'architettura dell'8088, alla sua organizzazione di memoria, alle modalità d'indirizzamento e alle istruzioni. Il Paragrafo C.5 descrive l'assemblatore impiegato in questa appendice, reperibile nel CD-ROM allegato al testo, e noi ci atterremo alla notazione usata da questo assemblatore. I lettori abituati alla notazione del linguaggio assemblativo dell'8088 dovrebbero tener presente che altri assemblatori usano notazioni diverse. Il Paragrafo C.6 illustra lo strumento interpret/tracer/debugger che assiste il programmatore nel debugging dei programmi e che si trova ugualmente nel materiale di supporto al testo. Il Paragrafo C.7 concerne l'installazione degli strumenti e la guida al loro utilizzo. Il Paragrafo C.8 contiene alcuni programmi, esempi, eser-

cizi e soluzioni. Il Paragrafo C.9 è una digressione su alcune questioni implementative, sui bachi e sulle limitazioni del materiale fornito.

## C.1 Panoramica

Cominciamo la nostra visita guidata alla programmazione in linguaggio assemblativo spendendo qualche parola sul linguaggio stesso e fornendo poi qualche esempio.

### C.1.1 Linguaggio assemblativo

Ogni assemblatore utilizza **nomi mnemonici**, cioè delle abbreviazioni quali ADD, SUB e MUL per denotare alcune entità e renderle più facili da ricordare: in questo caso addizione, sottrazione e moltiplicazione. Inoltre, gli assemblatori usano **nomi simbolici** per variabili e costanti, ed **etichette** per le istruzioni e per gli indirizzi di memoria. Molti assemblatori supportano poi alcune **direttive** che non vengono tradotte in istruzioni ISA, ma rappresentano comandi impartiti all'assemblatore per guidarne l'attività.

L'**assemblatore** (*assembler*) è un programma che riceve in ingresso un file contenente un programma in linguaggio assemblativo, e genera un file contenente un **programma binario** adatto all'esecuzione vera e propria, cioè pronto per essere eseguito dall'hardware. Tuttavia, i principianti della programmazione in linguaggio assemblativo commetteranno spesso degli errori che causano l'interruzione improvvisa dell'esecuzione, senza alcuna indicazione circa il problema che si è verificato.

Per semplificare la vita dei principianti, è possibile far eseguire il programma binario non all'hardware vero e proprio, ma a un simulatore che esegue un'istruzione per volta e visualizza un resoconto dettagliato della sua attività. Così facendo si semplifica notevolmente l'attività di debugging. I programmi eseguiti dai simulatori sono ovviamente molto lenti, ma ciò non costituisce un problema se l'obiettivo è quello di apprendere la programmazione in linguaggio assemblativo. Questa appendice si basa su vari strumenti di sviluppo tra cui un simulatore chiamato **interprete**, o **tracer**, poiché interpreta e traccia l'esecuzione del programma binario un'istruzione alla volta. I termini "simulatore", "interprete" e "tracer" saranno usati in modo del tutto intercambiabile nel corso della trattazione. In genere parleremo di "interprete" quando ci riferiremo alla sola esecuzione di un programma, mentre parleremo di "tracer" quando ci riferiremo allo strumento di debugging, ma si tratta sempre dello stesso programma.

### C.1.2 Breve programma in linguaggio assemblativo

Concretizziamo queste idee astratte considerando il programma e il tracer della Figura C.1. La figura illustra una schermata del tracer e un breve programma nel linguaggio assemblativo dell'8088. I numeri che si trovano dopo i punti esclamativi sono i numeri delle righe di codice e fanno riferimento a parti del programma. Il CD allegato al testo contiene una copia di questo programma nel file *HelloWorld.s* all'interno della directory *examples*. Questo programma ha estensione *.s*, come tutti i programmi assemblativi trattati in questa appendice, a indicare il fatto che si tratta del sorgente di un programma in linguaggio assemblativo. La schermata del tracer, mostrata nella Figura C.1(b), contiene sette finestre, ciascuna delle quali riporta alcune informazioni sullo stato del programma in esecuzione.

Esaminiamo brevemente la Figura C.1(b). La finestra in alto a sinistra mostra il contenuto del processore, ovvero il valore corrente dei registri di segmento CS, DS, SS ed ES, dei registri aritmetici AH, AL, AX e di altri.

La finestra in alto al centro rappresenta lo stack, l'area di memoria per le variabili temporanee.

La finestra in alto a destra contiene un frammento del programma in linguaggio assemblativo, e la freccia indica l'istruzione man mano in esecuzione. La comodità del tracer sta nel fatto che basta premere il tasto Invio per eseguire un'istruzione e aggiornare simultaneamente tutte le finestre, perciò è possibile eseguire il programma un'istruzione per volta.

Al di sotto della finestra di sinistra ce n'è una contenente lo stack delle chiamate di subroutine (in questo caso è vuota). Ancora più in basso vediamo i comandi del tracer. A destra di queste due finestre si trova la finestra per l'input/output e per i messaggi d'errore.

La finestra al fondo della figura mostra una porzione della memoria. Più avanti descriveremo queste finestre in maggior dettaglio, ma dovrebbe essere già chiara l'idea fondamentale: il tracer mostra il sorgente del programma, i registri di macchina e varie informazioni sullo stato del programma in esecuzione. Le informazioni vengono aggiornate dopo l'esecuzione di ogni istruzione, permettendo all'utente di osservare il funzionamento del programma in ogni dettaglio.

<pre> _EXIT = 1          !1 _WRITE = 4          !2 _STDOUT = 1         !3 .SECT TEXT          !4 start:             !5     MOV CX,de-hw    !6     PUSH CX          !7     PUSH hw          !8     PUSH _STDOUT     !9     PUSH _WRITE      !10     SYS              !11     ADD SP, 8         !12     SUB CX,AX        !13     PUSH CX          !14     PUSH _EXIT       !15     SYS              !16 .SECT DATA          !17 hw:                !18 .ASCII "Hello World\n" !19 de: .BYTE 0         !20 </pre>	<pre> CS: 00   DS=SS=ES: 002 AH:00  AL:0c  AX: 12 BH:00  BL:00  BX:  0 CH:00  CL:0c  CX: 12 DH:00  DL:00  DX:  0 SP: 7id8  SF O D S Z C =&gt;0004 BP: 0000  CC -&gt; p -- 0001 =&gt; SI: 0000  IP:000c:PC 0000 DI: 0000  start + 7  000c  MOV CX,de-hw    !6 PUSH CX          !7 PUSH HW          !8 PUSH _STDOUT     !9 PUSH _WRITE      !10 SYS              !11 ADD SP,8         !12 SUB CX,AX        !13 PUSH CX          !14 PUSH _EXIT       !15 SYS              !16 </pre>	<pre> E I  hw &gt; Hello World\n  hw + 0 = 0000: 48 65 6c 6f 20 57 6f Hello World 25928 </pre>
--	--	--

(a)

(b)

Figura C.1 (a) Programma in linguaggio assemblativo. (b) Schermata del tracer corrispondente alla sua esecuzione.

## C.2 Processore 8088

Tutti i processori sono provvisti di uno stato interno contenente alcune informazioni cruciali per il loro funzionamento. A tal fine, il processore dispone di un insieme di **registri** dove memorizzare ed elaborare queste informazioni. Probabilmente il registro più importante è il **PC** (**program counter**), contenente la locazione di memoria della successiva istruzione da eseguire, ovvero il suo **indirizzo**. A volte, questo registro è chiamato **IP** (**instruction pointer**). Le istruzioni da eseguire si trovano in una parte della memoria detta **segmento di codice**. La memoria principale dell'8088 può superare di poco il megabyte, ma il segmento di codice corrente è di soli 64 KB. Il registro CS della Figura C.1 indica l'inizio del segmento di codice all'interno della memoria (da 1 MB). È possibile attivare un nuovo segmento di codice semplicemente cambiando il contenuto del registro CS. Allo stesso modo, esiste anche un segmento dati di 64 KB contenente i dati del programma. Il registro DS punta alla prima locazione del segmento dati e anch'esso può essere modificato per accedere a dati esterni al segmento dati corrente. I registri CS e DS sono necessari perché l'8088 ha registri di 16 bit che non possono memorizzare direttamente i 20 bit di un indirizzo di una memoria di 1 MB. È questa la ragione per cui furono introdotti i registri di segmento.

Gli altri registri contengono i dati e i puntatori ai dati che si trovano nella memoria principale. I programmi in linguaggio assemblativo possono accedere direttamente a questi registri. Oltre ai registri, il processore contiene tutta la circuiteria per l'esecuzione delle istruzioni, ma questa è accessibile al programmatore solo attraverso la dichiarazione d'istruzioni.

### C.2.1 Ciclo del processore

L'attività dell'8088 (e degli altri calcolatori) consiste nell'esecuzione d'istruzioni in rapida successione. L'esecuzione di una singola istruzione può essere scomposta nelle fasi seguenti:

1. fetch dell'istruzione dalla memoria (in particolare dal segmento di codice) tramite il PC
2. incremento del program counter
3. decodifica dell'istruzione prelevata
4. fetch dei dati necessari dalla memoria e/o dai registri del processore
5. svolgimento dell'istruzione
6. memorizzazione dei risultati dell'istruzione in memoria e/o nei registri
7. ritorno alla fase 1 per l'avvio dell'istruzione successiva.

L'esecuzione di un'istruzione è simile a quella di un programma in miniatura. Infatti, alcune macchine usano davvero un programma molto piccolo, il **microprogramma**, per l'esecuzione delle loro istruzioni. I microprogrammi sono descritti ampiamente nel Capitolo 4.

Dal punto di vista del programmatore in linguaggio assemblativo, l'8088 ha 14 registri che costituiscono in un certo senso lo spazio di lavoro a disposizione delle istruzioni, anche se i dati ivi memorizzati vengono modificati molto spesso e non sono quindi molto adatti a contenere risultati definitivi. La Figura C.2 dà una visione d'insieme dei 14 registri molto

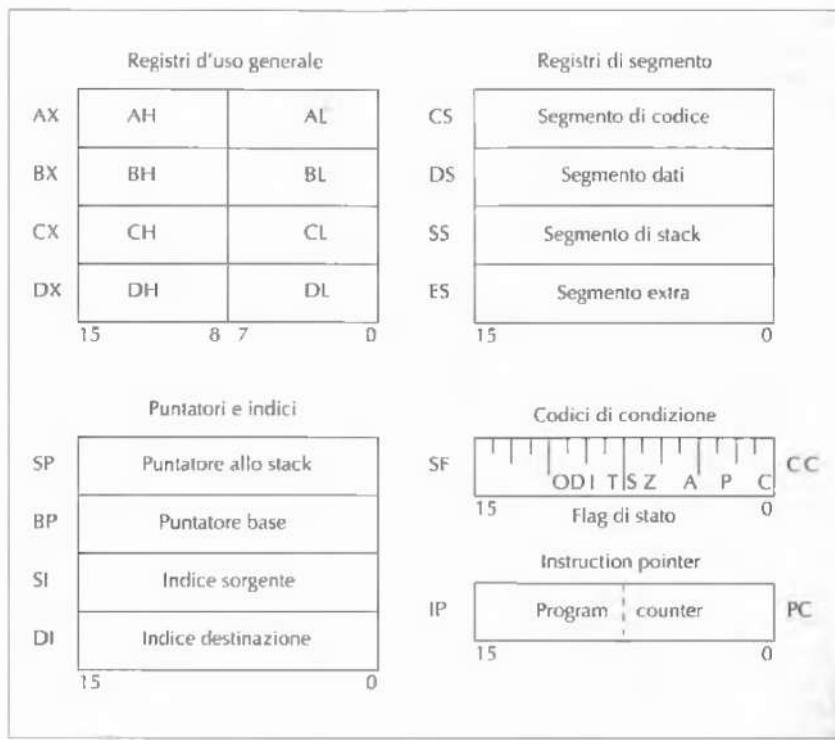


Figura C.2 Registri dell'8088.

simile alla finestra dei registri del tracer mostrata nella Figura C.1. Infatti le due figure rappresentano la stessa informazione.

I registri dell'8088 sono ampi 16 bit e svolgono tutte funzioni diverse. Alcuni di loro però condividono alcune caratteristiche, perciò è possibile individuare i gruppi della Figura C.2. Esaminiamo ciascun gruppo separatamente.

### C.2.2 Registri d'uso generale

I registri del primo gruppo, AX, BX, CX e DX, sono **d'uso generale**. Il primo di questi, AX, si chiama **registro accumulatore** (o semplicemente **accumulatore**) ed è usato per accogliere il risultato dell'elaborazione e funge da destinazione di un gran numero d'istruzioni. Anche se molte istruzioni possono specificare un registro qualsiasi per ospitare il loro risultato, altre definiscono implicitamente AX come destinazione del risultato dell'operazione, come fa per esempio la moltiplicazione.

Il secondo registro del gruppo è BX, il **registro base**. BX può essere usato al posto di AX per molte finalità, ma ha anche una funzionalità che AX non ha: è possibile memoriz-

zare in BX un indirizzo di memoria e poi eseguire un'istruzione il cui operando proviene dall'indirizzo di memoria contenuto in BX. Detto altrimenti, BX può contenere un puntatore alla memoria, ma AX no. Vediamo questa distinzione mediante due istruzioni. Per prima cosa scriviamo

`MOV AX, BX`

che copia in AX il contenuto di BX. Poi scriviamo

`MOV AX, (BX)`

che copia in AX il contenuto della parola di memoria che si trova all'indirizzo specificato da BX. Nel primo esempio BX contiene l'operando sorgente; nel secondo caso invece punta all'operando sorgente. In entrambi gli esempi l'istruzione MOV contiene un operando sorgente e uno destinazione, e la destinazione è indicata prima della sorgente.

Il registro CX è il **registro contatore**. È usato per diversi compiti, e in particolare come contatore dei cicli. Viene decrementato automaticamente dall'istruzione LOOP e i cicli terminano quando CX raggiunge il valore zero.

Il quarto registro d'uso generale è DX, il **registro dati**, usato congiuntamente ad AX per contenere le istruzioni lunghe due parole (32 bit). In tal caso DX contiene i 16 bit più significativi e AX gli altri. Generalmente, gli interi di 32 bit si dicono **long**, mentre il termine **double** è riservato ai valori in virgola mobile di 64 bit, benché qualcuno usi il termine "double" per riferirsi agli interi di 32 bit. In queste pagine non ci sarà alcuna ambiguità per il semplice fatto che non trattiamo i numeri in virgola mobile.

Tutti i registri d'uso generale possono essere visti come registri di 16 bit o come copie di registri di 8 bit. In tal modo l'8088 dispone esattamente di 8 registri di 8 bit utilizzabili per le istruzioni che trattano byte o caratteri. Nessun altro registro può essere suddiviso in due da 8 bit. È importante capire che AL e AH sono semplicemente i nomi delle due metà di AX. Quando si carica un nuovo valore in AX, AL e AH sono entrambi modificati per contenere rispettivamente la metà inferiore e la metà superiore del numero di 16 bit inserito in AX. Per comprendere la relazione tra AX, AH e AL si consideri l'istruzione

`MOV AX, 258`

che scrive il valore decimale 258 nel registro AX. Dopo l'esecuzione dell'istruzione, il registro di un byte AH contiene il valore 1 e il registro di un byte AL contiene il numero 2. Infatti, i 16 bit della rappresentazione binaria di 258 sono

00000001 00000010.

Se l'istruzione è seguita dall'istruzione di somma tra byte

`ADDB AH, AL`

allora il registro AH viene incrementato con il valore di AL (2) e quindi ora vale 3. L'effetto di questa azione è che ora il registro AX contiene il valore 770, equivalente a

00000011 00000010 (binario), ovvero 03 02 (esadecimale).

Gli otto registri da un byte sono quasi completamente intercambiabili, con l'eccezione che **AL** contiene sempre uno degli operandi dell'istruzione **MULB** ed è la destinazione implicita di questa operazione, insieme ad **AH**. Anche **DIVB** usa la coppia **AH:AL** per contenere il dividendo. Il byte meno significativo del registro contatore, **CL**, può essere usato per contenere il numero di cicli delle istruzioni di rotazione e scorrimento.

Il secondo esempio del Paragrafo C.8 mostra le proprietà dei registri d'uso generale attraverso la presentazione del programma *GenReg.s*.

### C.2.3 Registri puntatore

Il secondo gruppo di registri contiene i **registri puntatore** e i **registri indice**. Tra questi, il più importante è il **puntatore allo stack** denotato con **SP**. Uno stack è un segmento di memoria contenente informazioni sul contesto del programma in esecuzione. All'atto di una chiamata di procedura, una parte dello stack viene riservata alla memorizzazione delle sue variabili locali, dell'indirizzo dell'istruzione da eseguire dopo il suo completamento, e di altre informazioni di controllo. La parte di stack che riguarda una procedura si chiama il suo **record d'attivazione** (*stack frame*). Quando una procedura ne chiama un'altra, viene allocato un altro record d'attivazione di seguito a quelli già presenti. Pur non essendo una regola obbligatoria, in genere gli stack crescono verso il basso, dagli indirizzi maggiori verso i minori. Ciò nondimeno, l'ultimo indirizzo occupato dallo stack si chiama sempre cima dello stack.

Oltre a ospitare le variabili locali, gli stack possono servire anche per memorizzare i risultati temporanei. L'8088 dispone dell'istruzione **PUSH** per impilare una parola di 16 bit in cima allo stack. L'istruzione comincia con il decrementare **SP** di 2, per poi memorizzare il suo operando al nuovo indirizzo puntato da **SP**. Analogamente, l'istruzione **POP** rimuove una parola di 16 bit dalla cima dello stack prelevando il valore che si trova in cima e incrementando **SP** di 2. Il registro **SP** punta sempre alla cima dello stack e viene modificato dalle istruzioni **PUSH**, **POP** e **CALL**; in particolare viene decrementato da **PUSH** e **CALL**, e incrementato da **POP**.

Il registro successivo è **BP**, il **puntatore base**, che in genere punta a una locazione dello stack. A differenza di **SP**, che punta sempre alla cima dello stack, **BP** può puntare a una sua locazione qualsiasi. Nella pratica viene usato comunemente per puntare all'inizio del record d'attivazione della procedura corrente, per raggiungere le sue variabili locali. Dunque **BP** punta spesso alla base del record d'attivazione corrente (al suo indirizzo più grande) e **SP** alla sua cima (al suo indirizzo più piccolo), perciò questi due registri delimitano il record d'attivazione.

Appartengono allo stesso gruppo i due registri indice: **SI**, **indice sorgente** e **DI**, **indice destinazione**. Questi registri sono usati spesso in combinazione con **BP** per riferirsi ai dati dello stack, o con **BX** per localizzare i dati in memoria. Per una loro più ampia trattazione rimandiamo al paragrafo sulle modalità d'indirizzamento.

Uno dei registri più importanti, tanto da costituire un gruppo a sé, è l'**instruction pointer**, l'equivalente Intel del program counter (**PC**). Questo registro non viene usato direttamente dalle istruzioni, bensì contiene un indirizzo di memoria appartenente al segmento di codice del programma. Il ciclo del processore comincia con il fetch dell'istruzione puntata da **PC**, dopodiché questo viene incrementato prima del completamento dell'istruzione in esecuzione. Così facendo questo registro punta sempre all'istruzione che segue quella corrente.

**Il registro di flag, o dei codici di condizione.** è in pratica un insieme di registri da 1 bit. Alcuni di questi bit sono impostati dalle istruzioni aritmetiche e rappresentano il loro risultato, nei modi seguenti:

- Z – il risultato è zero
- S – il risultato è negativo (bit di segno)
- V – il risultato ha causato un overflow
- C – il risultato ha generato un riporto
- A – riporto ausiliario (oltre il bit 3)
- P – parità del risultato.

Gli altri bit del registro controllano certi aspetti dell'attività del processore. Il bit **I** attiva gli interrupt. Il bit **T** abilita la modalità di tracing, usata per il debugging, e il bit **D** controlla la direzione delle operazioni su stringhe. Non tutti i 16 bit del registro sono utilizzati; quelli inutilizzati sono cablati al valore zero.

Segue poi il gruppo dei quattro **registri di segmento**. Ricordiamo che i segmenti dello stack, dei dati e delle istruzioni risiedono tutti in memoria principale, ma spesso in posti diversi. I registri di segmento servono alla gestione di queste diverse porzioni della memoria che prendono il nome di **segmenti**. Questi registri comprendono **CS** per il segmento di codice, **DS** per il segmento dati, **SS** per il segmento di stack ed **ES** per il segmento extra. Il loro valore resta costante per la maggior parte del tempo. In quasi tutte le implementazioni, il segmento dati e il segmento di stack usano la stessa porzione di memoria, ma i dati occupano la sua base e lo stack è allocato alla sua cima. Approfondiremo lo studio di questi registri nel Paragrafo C.3.1.

## C.3 Memoria e indirizzamento

L'8088 ha un'organizzazione di memoria abbastanza grossolana a causa della combinazione infelice di una memoria di 1 MB e registri di 16 bit. La memoria di 1 MB richiede 20 bit per la rappresentazione di un suo indirizzo e dunque è impossibile memorizzare un puntatore alla memoria nei registri di 16 bit. Per aggirare il problema, la memoria è stata organizzata in segmenti di 64 KB e così è possibile rappresentare un indirizzo all'interno di un segmento con soli 16 bit. Vediamo l'architettura di memoria dell'8088 in maggior dettaglio.

### C.3.1 Organizzazione di memoria e segmenti

La memoria dell'8088 consiste in una semplice successione di byte indirizzabili, contenente istruzioni, dati e stack. L'8088 usa il concetto di **segmento** per distinguere le porzioni di memoria che assolvono a compiti diversi. Ogni segmento dell'8088 è costituito da 65.536 byte consecutivi. Ne esistono quattro:

1. il segmento di codice
2. il segmento dati
3. il segmento di stack
4. il segmento extra.

Il primo contiene le istruzioni del programma. Il contenuto del registro **PC** viene interpretato sempre come un indirizzo di memoria appartenente a questo segmento. Il valore 0 del **PC** sta a indicare l'indirizzo minore del segmento di codice, *non* l'indirizzo zero della memoria. Il segmento dati contiene i dati, inizializzati e non, del programma. Quando **BX** è usato come puntatore si riferisce proprio a questo segmento. Il segmento di stack contiene le variabili locali e i risultati intermedi impilati sullo stack. Gli indirizzi di **SP** e **BP** appartengono sempre a questo segmento. Il segmento extra è un segmento di riserva che può essere allocato in memoria ovunque sia necessario.

A ciascuno di questi segmenti corrisponde un registro di segmento: si tratta dei registri di 16 bit **CS**, **DS**, **SS** ed **ES**. L'indirizzo iniziale di un segmento si ottiene estendendo i 16 bit del registro corrispondente concatenandogli quattro zeri sulla destra (nelle posizioni meno significative), producendo così un intero senza segno di 20 bit. Ciò vuol dire che i registri di segmento indicano sempre indirizzi multipli di 16 nello spazio degli indirizzi di 20 bit. Ogni registro di segmento punta alla base del proprio segmento. Gli indirizzi all'interno del segmento si costruiscono convertendo i 16 bit del registro nell'indirizzo fisico di 20 bit e sommandogli l'offset. Nella pratica, un indirizzo assoluto di memoria si calcola moltiplicando il registro di segmento per 16 e sommandogli poi l'offset. Per esempio, se **DS** vale 7 e **BX** vale 12, l'indirizzo indicato da **BX** è  $7 \times 16 + 12 = 124$ . In altre parole, l'indirizzo binario di 20 bit indotto da **DS** = 7 è 00000000000011100000. La somma dell'offset di 16 bit 0000000000001100 (12 in decimale) all'indirizzo base del segmento produce l'indirizzo fisico 000000000000111100 (124 in decimale).

Ogni riferimento alla memoria richiede l'intervento di un registro di segmento per la costruzione dell'indirizzo fisico. Se un'istruzione contiene un indirizzo e non specifica alcun registro di segmento, allora è interpretata automaticamente come riferimento al segmento dati e si usa **DS** come base dell'offset. L'indirizzo fisico si ottiene sommando questa base all'indirizzo contenuto nell'istruzione. L'indirizzo fisico della successiva istruzione di codice si ottiene facendo scorrere il contenuto di **CS** di quattro posizioni a sinistra e sommando il valore del program counter. In altre parole, viene prima calcolato l'indirizzo di 20 bit indotto dai 16 bit del registro **CS**, poi viene effettuata la somma con i 16 bit del **PC** per formare l'indirizzo di memoria assoluto di 20 bit.

Il segmento di stack è costituito da parole di 2 byte, perciò il puntatore allo stack, **SP**, contiene sempre un numero pari. Lo stack viene riempito a partire dagli indirizzi più grandi in senso decrescente. L'istruzione **PUSH** decremente il puntatore allo stack di 2 e memorizza l'operando nell'indirizzo di memoria calcolato mediante **SS** e **SP**. Il comando **POP** recupera il valore in cima e incrementa **SP** di 2. Gli indirizzi del segmento di stack al di sotto di quello indicato da **SP** si considerano liberi. La rimozione di una porzione dello stack si consegna con un semplice incremento di **SP**. Nella pratica **DS** e **SS** valgono sempre lo stesso valore, così si può usare un solo puntatore di 16 bit per accedere al segmento condiviso da stack e dati. Se **DS** e **SS** fossero stati diversi, si sarebbe reso necessario un diciassettesimo bit per distinguere tra puntatori al segmento dati e puntatori al segmento di stack. A posteriori, si può dire che la stessa definizione di un segmento di stack separato dai dati si è rivelata un errore.

Se gli indirizzi dei quattro segmenti sono molto distanti tra loro, allora i segmenti risultanti saranno disgiunti, ma ciò non è necessario; per esempio, se c'è poca memoria disponibile è possibile che i segmenti si sovrappongano almeno in parte. Al termine della com-

pilazione, la dimensione del codice del programma è nota. Una scelta efficiente prevede di far cominciare il segmento condiviso dallo stack e dai dati al primo multiplo di 16 che si trova dopo l'ultima istruzione. Questa scelta implica che i segmenti del codice e dei dati non useranno mai gli stessi indirizzi fisici.

### C.3.2 Indirizzamento

Quasi tutte le istruzioni richiedono dati provenienti dalla memoria o dagli altri registri. L'8088 mette a disposizione un ventaglio di modalità d'indirizzamento abbastanza versatile per far riferimento ai dati. Molte istruzioni contengono due operandi, chiamati in genere **destinazione** e **sorgente**. Si pensi, per esempio, all'istruzione di copia o di somma:

```
MOV AX, BX    O ADD CX, 20
```

Il primo operando di queste istruzioni è la destinazione, il secondo la sorgente (l'ordine scelto da Intel è arbitrario: sarebbe stata lecita anche la scelta opposta). È evidente che la destinazione deve essere un **valore di sinistra dell'assegnamento** (*left value*), cioè deve essere una locazione in cui è lecito memorizzare un valore. Ciò implica che le costanti possono fungere da sorgenti, ma non da destinazioni.

Il progetto originale dell'8088 richiedeva che almeno un operando di ogni istruzione a due operandi fosse un registro. Questa scelta serviva per distinguere le **istruzioni di una parola** dalle **istruzioni di un byte** mediante il semplice controllo del registro indirizzato, a seconda che si trattasse di un **registro di una parola** o di uno **di un byte**. Nella prima versione del processore quest'idea veniva rispettata così rigorosamente tanto da non permettere neanche di impilare una costante sullo stack, dato che in questo caso nessuno dei due operandi coinvolge un registro. Le versioni successive sono meno rigorose, tuttavia l'idea ha continuato ugualmente a influenzare il progetto del processore. In alcuni casi uno dei due operandi viene omesso. Per esempio, nel caso dell'istruzione **MULB**, solo il registro **AX** può fungere da destinazione.

Esistono anche alcune istruzioni a un operando, come gli incrementi, gli scorimenti, le negazioni, ecc. In questi casi non c'è vincolo sui registri e la differenza tra operazioni di una parola o di un byte deve essere desunta dall'opcode (cioè dal tipo d'istruzione).

L'8088 supporta quattro tipi di dati fondamentali: i **byte**, le **parole** di 2 byte, il tipo **long** di 4 byte e i **decimali binari** (*Binary Coded Decimals*, BCD), che impacchettano due cifre decimali all'interno di una parola. Quest'ultimo tipo non è supportato dall'interprete.

Un indirizzo di memoria si riferisce sempre a un byte, ma in caso di riferimento a una parola o a un dato long ci si riferisce implicitamente anche ai byte che seguono quello indicato. La parola all'indirizzo 20 comprende le locazioni di memoria 20 e 21. Il dato long all'indirizzo 24 occupa le locazioni da 24 a 27. L'8088 assume l'ordinamento **little endian**, cioè la parte meno significativa di una parola è memorizzata all'indirizzo minore. All'interno del segmento stack, le parole devono essere allocate in corrispondenza degli indirizzi pari. La coppia di registri **DX:AX** è l'unica messa a disposizione dal processore per i dati long (**AX** contiene la parola meno significativa).

La tabella della Figura C.3 fornisce una visione d'insieme delle modalità d'indirizzamento dell'8088. Diamone un breve resoconto. La prima riga della tabella elenca i registri utilizzabili come operandi di quasi tutte le istruzioni, come sorgente o come destinazione. Ci sono otto registri di una parola e otto di un byte.

Modalità	Operando	Esempi
<b>Indirizzamento a registro</b>		
Registro da un byte	Registro da un byte	AH, AL, BH, BL, CH, CL, DH, DL
Registro da una parola	Registro da una parola	AX, BX, CX, DX, SP, BP, SI, DI
<b>Indirizzamento del segmento dati</b>		
Indirizzo diretto	L'indirizzo segue l'opcode	(#)
A registro indiretto	L'indirizzo è nel registro	(SI), (DI), (BX)
A registro con spiazzamento	L'indirizzo è dato da registro + spiazz.	A(SI), #DI, #BX
A registro con indice	L'indirizzo è BX + SI/DI	(BX)(SI), (BX)(DI)
A registro con indice e spiazzamento	BX + SI/DI + spiazzamento	#(BX)(SI), #(BX)(DI)
<b>Indirizzamento del segmento di stack</b>		
Indiretto a puntatore base	L'indirizzo è nel registro	(BP)
A puntatore base con spiazzamento	L'indirizzo è BP + spiazz.	#(BP)
A puntatore base con indice	L'indirizzo è BP + SI/DI	(BP)(SI), (BP)(DI)
A puntatore base con indice e spiazz.	BP + SI/DI + spiazzamento	#(BP)(SI), #(BP)(DI)
<b>Dati immediati</b>		
byte/parola immediato/a	I dati sono parte dell'istruzione	#
<b>Indirizzamento implicito</b>		
Istruzione di push/pop	Indirizzo indiretto (SP)	PUSH, POP, PUSHF, POPF
Flag di load/store	registro dei flag di stato	LAHF, STC, CLC, CMC
Traduzione XLAT	AL, BX	XLAT
Istruzioni reiterate su stringhe	(SI), (DI), (CX)	MOVS, CMPS, SCAS
Istruzioni di input/out	AX, AL	IN #, OUT #
Conversioni di byte o parole	AL, AX, DX	CBW, CWD

Figura C.3 Modalità d'indirizzamento degli operandi. Il simbolo # indica un valore numerico o un'etichetta.

La seconda riga riguarda le modalità d'indirizzamento del segmento dati. Questi indirizzi sono sempre racchiusi tra parentesi a indicare che il contenuto va inteso come indirizzo e non come valore. La **modalità d'indirizzamento** più semplice è quella **diretta**, mediante cui l'istruzione contiene l'indirizzo dei dati nell'operando stesso. Per esempio:

ADD CX, (20)

specificava che bisogna sommare a CX il contenuto della parola di memoria agli indirizzi 20 e 21. Nel linguaggio assemblativo si usa spesso rappresentare le locazioni di memoria mediante etichette e non con il loro valore numerico; la conversione da etichette a numeri viene effettuata dall'assemblatore. Anche la destinazione delle istruzioni CALL e JMP può essere specificata da un'etichetta. Le parentesi che racchiudono l'etichetta sono fondamentali (per l'assemblatore che stiamo descrivendo) perché anche

ADD CX, 20

è un'istruzione valida, ma produce la somma di CX con la costante 20, non con il contenuto della locazione 20. Il simbolo # usato nella Figura C.3 indica le costanti numeriche, le etichette o le espressioni costanti contenenti un'etichetta.

Nell'**indirizzamento a registro indiretto**, l'indirizzo dell'operando viene memorizzato in uno dei registri BX, SI o DI. In tutti e tre i casi si tratta di un operando che fa riferimento al segmento dati. È possibile anche far precedere un registro da un simbolo di costante, nel qual caso l'indirizzo si ottiene commandando la costante al registro. Questo tipo d'in-

dirizzamento si chiama **indirizzamento con spiazzamento** ed è particolarmente utile per il trattamento degli array. Per esempio, se SI contiene il valore 5, è possibile caricare in AL il quinto carattere della stringa *FORMAT* tramite

MOV B AL, FORMAT(SI).

L'intera stringa può essere scandita incrementando o decrementando il registro a ogni passo. Se si usano gli operandi di una parola, bisogna avere l'accortezza di modificare il registro di 2 unità alla volta.

È possibile anche memorizzare la base dell'array (cioè il suo indirizzo minore) nel registro BX e utilizzare il registro SI o DI per scadirlo. Si parla in tal caso d'**indirizzamento a registro con indice**. Per esempio

PUSH (BX)(DI)

preleva il contenuto della locazione del segmento dati il cui indirizzo è la somma dei registri BX e DI, e lo impila in cima allo stack. Gli ultimi due tipi d'indirizzamento possono essere combinati per ottenere l'**indirizzamento a registro con indice e spiazzamento**, come in

NOT 20(BX)(DI)

che fa il complemento della parola di memoria che si trova agli indirizzi BX + DI + 20 e seguente.

Tutte le modalità d'indirizzamento indiretto del segmento dati si possono usare anche per il segmento di stack, ma in tal caso si usa come puntatore base BP al posto del registro BX. Così facendo, (BP) è la sola modalità d'indirizzamento a registro indiretto disponibile per lo stack, ma esistono altre modalità via via più complicate che culminano nella modalità indiretta a puntatore base con indice e spiazzamento, per esempio -1(BP)(SI). Queste modalità sono utili per indirizzare le variabili locali e i parametri di funzione, memorizzati presso gli indirizzi di stack delle subroutine. Questo sistema è descritto più a fondo nel Paragrafo C.4.5.

Tutti gli indirizzi che rispettano le modalità d'indirizzamento fin qui trattate possono fungere da sorgenti o da destinazioni delle operazioni e si dicono **indirizzi effettivi**. Le modalità d'indirizzamento delle rimanenti due righe non possono essere usate per le destinazioni e non danno luogo a indirizzi effettivi. Possono servire solo per specificare sorgenti.

La modalità d'indirizzamento in cui l'operando dell'istruzione è una costante di un byte o di una parola si dice **indirizzamento immediato**. Per esempio

CMP AX, 50

confronta AX con la costante 50 e memorizza l'esito del confronto nel registro dei flag.

Infine, alcune istruzioni usano l'**indirizzamento implicito**, secondo il quale l'operando o gli operandi dell'istruzione sono specificati implicitamente dall'istruzione stessa. Per esempio l'istruzione

PUSH AX

impila il contenuto di AX in cima allo stack mediante il decremento di SP e la copia di AX nella nuova locazione puntata da SP. Il registro puntatore allo stack non viene indicato dall'istruzione, ma il solo fatto che si tratti di un'istruzione PUSH implica l'uso di SP. Anche le istruzioni di manipolazione dei flag usano implicitamente il registro dei flag senza nominarlo, e molte altre istruzioni usano operandi impliciti.

L'8088 dispone d'istruzioni speciali per il trasferimento (**MOV**), il confronto (**CMPS**) e la scansione (**SCAS**) di stringhe. Queste istruzioni sulle stringhe comportano la modifica automatica dei registri indice **SI** e **DI** al termine delle operazioni corrispondenti. Questo comportamento si dice **autoincremento o autodecremento**. **SI** e **DI** vengono modificati a seconda del valore assunto dal **flag della modalità d'incremento** (*direction flag*) contenuto nel registro dei flag: se vale 0 allora si ha un incremento, altrimenti un decremento. La modifica è di un'unità alla volta nel caso d'istruzioni di un byte e di 2 per le istruzioni di una parola. In un certo senso anche il puntatore allo stack segue le modalità di autoincremento e autodecremento: viene decrementato di 2 all'inizio di una **PUSH** e incrementato di 2 alla fine di una **POP**.

## C.4 Istruzioni dell'8088

Il fulcro di ogni calcolatore è l'insieme d'istruzioni che può eseguire. La comprensione profonda di un calcolatore passa per la comprensione dell'insieme delle sue istruzioni. I paragrafi seguenti sono dedicati all'esposizione delle istruzioni più importanti dell'8088. La Figura C.4 ne mostra alcune.

### C.4.1 Trasferimento, copia e aritmetica

Il primo gruppo contiene le istruzioni di copia e di trasferimento. L'istruzione di gran lunga più utilizzata è **MOV**, che specifica esplicitamente una sorgente e una destinazione. Se la sorgente è un registro, allora la destinazione può essere un indirizzo effettivo. Nella tabella delle istruzioni indichiamo con una *r* gli operandi registro e con una *e* gli indirizzi effettivi, dunque la precedente combinazione degli operandi della **MOV** si indica con *e* → *r*. Il verso della freccia segue la convenzione sintattica secondo cui si indicano prima le destinazioni e poi le sorgenti; dunque *e* → *r* vuol dire che un registro viene copiato in un indirizzo effettivo.

Si tratta solo del primo elemento della colonna *Operandi* in corrispondenza dell'istruzione **MOV**, ma è anche possibile avere per sorgente un indirizzo effettivo e per destinazione un registro, *r* ← *e*, oppure un dato immediato come sorgente e un indirizzo effettivo come destinazione, che indichiamo con *e* ← #. I dati immediati sono contrassegnati nella tabella con il *cancelloetto* (#). La *B* indicata tra parentesi dopo il nome dell'istruzione serve a ricordare che esiste sia l'istruzione **MOV** per i trasferimenti di parole, sia l'istruzione **MOV***B* per i trasferimenti di byte. Perciò la prima riga della tabella riguarda in realtà sei istruzioni diverse.

Le istruzioni di trasferimento non hanno alcun effetto sul registro dei codici di condizione e quindi le ultime quattro colonne sono contrassegnate con il simbolo “-”. Si noti che in realtà le istruzioni di trasferimento non trasferiscono i dati, ma li copiano, ovvero la sorgente non viene modificata come accadrebbe a seguito di un vero trasferimento.

La seconda istruzione della tabella è **XCHG**, che scambia il contenuto di un registro con il contenuto di un indirizzo effettivo. Il simbolo per lo scambio è ↔. Anche in questo caso esiste sia la versione per le parole, sia quella per i byte. La riga della tabella contiene quindi *r* ↔ *e* nella colonna *Operandi* in corrispondenza della riga di **XCHG**. L'istruzione successiva è **LEA**, che sta per Load Effective Address (“caricamento di un indirizzo effettivo”). Serve a calcolare il valore numerico dell'indirizzo effettivo e a memorizzarlo in un registro.

Abbreviazione	Descrizione	Operandi	Flag di stato			
			O	S	Z	C
MOV(B)	Trasferimento di parola o byte	<i>r</i> ← <i>e</i> , <i>e</i> ← <i>r</i> , <i>e</i> ← #	-	-	-	-
XCHG(B)	Scambio di parole	<i>r</i> ↔ <i>e</i>	-	-	-	-
LEA	Caricamento di un indirizzo fisico	<i>r</i> ← # <i>e</i>	-	-	-	-
PUSH	Push sullo stack	<i>e</i> , #	-	-	-	-
POP	Pop dallo stack	<i>e</i>	-	-	-	-
PUSHF	Push dei flag	-	-	-	-	-
POPF	Pop dei flag	-	-	-	-	-
XLAT	Traduzione di AL	-	-	-	-	-
ADD(B)	Somma di parole	<i>r</i> ← <i>e</i> , <i>e</i> ← <i>r</i> , <i>e</i> ← #	*	*	*	*
ADC(B)	Somma di parole con riporto	<i>r</i> ← <i>e</i> , <i>e</i> ← <i>r</i> , <i>e</i> ← #	*	*	*	*
SUB(B)	Sottrazione di parole	<i>r</i> ← <i>e</i> , <i>e</i> ← <i>r</i> , <i>e</i> ← #	*	*	*	*
SBB(B)	Sottrazione di parole con prestito	<i>r</i> ← <i>e</i> , <i>e</i> ← <i>r</i> , <i>e</i> ← #	*	*	*	*
IMUL(B)	Moltiplicazione con segno	<i>e</i>	*	U	U	*
MUL(B)	Moltiplicazione senza segno	<i>e</i>	*	U	U	*
IDIV(B)	Divisione con segno	<i>e</i>	U	U	U	U
DIV(B)	Divisione senza segno	<i>e</i>	U	U	U	U
CBW	Estensione da byte a parola	-	-	-	-	-
CWD	Estensione da parola a double	-	-	-	-	-
NEG(B)	Complemento binario	<i>e</i>	*	*	*	*
NOT(B)	Complemento logico	<i>e</i>	*	*	*	*
INC(B)	Incremento della destinazione	<i>e</i>	*	*	*	*
DEC(B)	Decremento della destinazione	<i>e</i>	*	*	*	*
AND(B)	AND	<i>e</i> ← <i>r</i> , <i>r</i> ← <i>e</i> , <i>e</i> ← #	0	*	*	0
OR(B)	OR	<i>e</i> ← <i>r</i> , <i>r</i> ← <i>e</i> , <i>e</i> ← #	0	*	*	0
XOR(B)	OR esclusivo	<i>e</i> ← <i>r</i> , <i>r</i> ← <i>e</i> , <i>e</i> ← #	0	*	*	0
SHR(B)	Scorrimento logico verso destra	<i>e</i> ← 1, <i>e</i> ← CL	*	*	*	*
SAR(B)	Scorrimento aritmetico verso destra	<i>e</i> ← 1, <i>e</i> ← CL	*	*	*	*
SAL(B) (=SHL(B))	Scorrimento logico verso sinistra	<i>e</i> ← 1, <i>e</i> ← CL	*	*	*	*
ROL(B)	Rotazione a sinistra	<i>e</i> ← 1, <i>e</i> ← CL	*	*	*	*
ROR(B)	Rotazione a destra	<i>e</i> ← 1, <i>e</i> ← CL	*	*	*	*
RCL(B)	Rotazione a sinistra con riporto	<i>e</i> ← 1, <i>e</i> ← CL	*	*	*	*
RCR(B)	Rotazione a destra con riporto	<i>e</i> ← 1, <i>e</i> ← CL	*	*	*	*
TEST(B)	Test degli operandi	<i>e</i> ← <i>r</i> , <i>e</i> ↔ #	0	*	*	0
CMP(B)	Confronto degli operandi	<i>e</i> ↔ <i>r</i> , <i>e</i> ↔ #	*	*	*	*
STD	Assegnazione flag modalità di incr. (↓)	-	*	*	*	*
CLD	Azzeramento flag modalità di incr. (↑)	-	*	*	*	*
STC	Assegnazione flag di riporto	-	-	-	-	1
CLC	Azzeramento flag di riporto	-	-	-	-	0
CMC	Complemento del riporto	-	*	*	*	*
LOOP	Salta all'indietro se decrementato CX ≥ 0	etichetta	-	-	-	-
LOOPZ LOOPE	Salta se Z=1 e DEC(CX) ≥ 0	etichetta	-	-	-	-
LOOPNZ LOOPNE	Salta se Z=0 e DEC(CX) ≥ 0	etichetta	-	-	-	-
REP REPZ REPNZ	Istruzione ripetitiva sulle stringhe	istruzioni sulle stringhe	-	-	-	-
MOV(B)	Trasf. di una stringa di una parola	-	-	-	-	-
LODS(B)	Caricam. di una stringa di parole	-	-	-	-	-
STOS(B)	Memorizz. di una stringa di parole	-	-	-	-	-
SCAS(B)	Scansione di una stringa di parole	-	*	*	*	*
CMPS(B)	Confronto tra stringhe di parole	-	*	*	*	*
JCC	Salto condizionato	etichetta	-	-	-	-
JMP	Salto all'etichetta	<i>e</i> , etichetta	-	-	-	-
CALL	Chiamata di subroutine	<i>e</i> , etichetta	-	-	-	-
RET	Ritorno da subroutine	-, #	-	-	-	-
SYS	Chiamata di sistema per trap	-	-	-	-	-

Figura C.4 Alcune delle istruzioni più importanti dell'8088.

Segue **PUSH** che impila operandi sullo stack. L'operando esplicito può essere una costante (# nella colonna *Operandi*) o un indirizzo effettivo (e nella colonna *Operandi*). C'è anche un operando implicito, **SP**, che non si evince dalla sintassi dell'istruzione. La semantica dell'istruzione è invece la seguente: decrementa **SP** di 2 e memorizza l'operando esplicito nella nuova locazione di memoria puntata da **SP**.

Troviamo poi l'istruzione **POP**, che rimuove un operando dallo stack e lo salva in un indirizzo effettivo. Anche le due istruzioni successive, **PUSHF** e **POPF**, hanno operandi impliciti, dato che effettuano il push e la pop del registro dei flag. Lo stesso vale per **XLAT**, che serve a caricare nel registro **AL** il byte che si trova all'indirizzo **AL + BX**. È un'istruzione molto utile per fare ricerche rapide nelle tabelle di dimensioni pari a 256 byte.

Le istruzioni **IN** e **OUT** sono definite dall'8088, ma non sono state implementate nell'interprete (e non sono elencate nelle Figura C.4). Si tratta infatti d'istruzioni di trasferimento da e verso i dispositivi di I/O. L'indirizzo implicito è sempre il registro **AX** e il secondo operando dell'istruzione è il numero di porta del registro di dispositivo desiderato.

Il secondo gruppo della tabella comprende le istruzioni di somma e sottrazione. Ciascuna di loro dispone delle stesse tre combinazioni di operandi di **MOV**: da indirizzo effettivo a registro, da registro a indirizzo effettivo e da costante a indirizzo effettivo. Quindi la colonna *Operandi* della tabella contiene  $r \leftarrow e$ ,  $e \leftarrow r$  e  $e \leftarrow #$ . Tutte le istruzioni di questo gruppo modificano i flag di stato (cioè i bit di overflow *O*, di segno *S*, di zero *Z* e di riporto *C*) in base al risultato dell'istruzione eseguita. Per esempio, *O* viene asserito se il risultato non può essere espresso correttamente con il numero di bit consentiti, e viene azzerato altrimenti. Se si prende il numero più grande rappresentabile con 16 bit, 0x7fff (32.767 in decimale), e lo si somma a se stesso, il risultato non può essere espresso con un numero di 16 bit con segno, così *O* viene asserito a indicare l'errore. Gli altri flag di stato sono gestiti in modo analogo. Indichiamo con un asterisco (\*) i flag che sono modificati dalle istruzioni. Le istruzioni **ADC** e **SBB** attingono al flag di riporto e lo usano come bit di riporto o di prestito generato dall'operazione precedente. Questa caratteristica è particolarmente utile per rappresentare gli interi di 32 bit (o ancora più lunghi) usando più parole consecutive. Tutte le istruzioni di somma o sottrazione esistono anche in versione per byte.

Il gruppo successivo contiene le istruzioni di moltiplicazione e divisione **MUL** e **DIV** con operandi interi con segno, e **MUL** e **DIV** per quelli senza segno. Le versioni per byte definiscono implicitamente come destinazione i registri accoppiati **AH:AL**, mentre le versioni per operandi di una parola usano come destinazione implicita la coppia di registri **DX:AX**. **DX** o **AH** vengono riscritti anche se il risultato della moltiplicazione occupa solo una parola o un byte. I bit di overflow e di riporto sono asseriti quando il risultato eccede la prima parola o byte; comunque la moltiplicazione può sempre andare a buon fine perché la destinazione (due parole o due byte) è abbastanza capiente per contenere il risultato. Il contenuto del flag *Z* e *S* è indefinito (*U*) dopo una moltiplicazione.

Anche la divisione usa come destinazione le combinazioni di registri **DX:AX** (per il resto) **AH:AL** (per il quoziente). Al termine di una divisione è indefinito il contenuto dei flag di resto, di overflow, di segno, e di zero. L'operazione causa una **trap** se il divisore è nullo o se non c'è abbastanza spazio per rappresentare il quoziente; la trap causa la terminazione del programma a meno che sia stata definita una routine per la gestione delle trap. Inoltre, conviene gestire via software il segno prima e dopo la divisione, perché l'8088 definisce il segno del resto in base a quello del dividendo, mentre dal punto di vista matematico il resto è sempre non negativo.

Le istruzioni **AAA** (ASCII Adjust for Addition) e **DAA** (Decimal Ajust for Addition) per le operazioni sui BCD non sono state implementate nell'interprete e quindi non sono presenti nella tabella.

#### C.4.2 Operazioni logiche, su bit e di scorrimento

Il blocco d'istruzioni seguente raggruppa le estensioni con segno, le negazioni, il complemento logico, l'incremento e il decremento. Le operazioni d'estensione con segno non hanno operandi espliciti, bensì operano sulle combinazioni di registri **DX:AX** o **AH:AL**. L'operando delle altre istruzioni può trovarsi invece presso qualsiasi indirizzo effettivo. Le operazioni **NEG**, **INC** e **DEC** hanno effetto sui flag in accordo con la loro semantica, anche se le operazioni d'incremento e decremento non modificano il bit di riporto e ciò è considerato da molti un errore di progettazione.

Seguono le istruzioni del gruppo logico, ciascuna dotata di due operandi e fedele alla semantica dell'operazione logica corrispondente. Il gruppo degli scorrimenti e delle rotazioni contiene istruzioni che hanno un indirizzo effettivo come destinazione, ma la loro sorgente può essere il registro di un byte **CL** o il numero 1. Gli scorrimenti modificano i quattro flag, mentre le rotazioni agiscono solo sui bit di riporto e di overflow. Il bit di riporto contiene sempre il bit che, a seguito di una rotazione o di uno scorrimento, "cade" fuori dal registro. Nelle rotazioni con riporto **RCR**, **RCL**, **RCRB** e **RCLB**, il bit di riporto e l'operando che si trova all'indirizzo effettivo costituiscono insieme una combinazione di 17 o di 9 bit per lo scorrimento circolare di registro, il che facilita gli scorrimenti e le rotazioni di più parole alla volta.

Il gruppo successivo d'istruzioni serve alla manipolazione dei bit dei flag. Queste operazioni sono utili soprattutto per la preparazione dei salti condizionati. Il rapporto tra i due operandi delle istruzioni di test e confronto, che non vengono modificati da queste operazioni, è indicato con il simbolo di doppia freccia (↔). L'operazione **TEST** comporta il calcolo dell'AND dei due operandi e, in base al risultato, imposta il bit zero e il bit di segno. Il risultato dell'AND non viene memorizzato da nessuna parte e gli operandi non vengono modificati. L'istruzione **CMP** invece calcola la differenza dei due operandi e modifica tutti i flag, in base al risultato del confronto. Il flag della modalità d'incremento, che specifica se le istruzioni sulle stringhe debbano incrementare o decrementare **SI** e **DI**, può essere assestito o azzerato dalle istruzioni **STD** e **CLD** rispettivamente.

L'8088 ha anche un **flag di parità** e uno **di riporto ausiliario**. Il primo indica la parità del risultato, e il secondo se si è verificato un overflow nel mezzo byte (4 bit) meno significativo della destinazione. Esistono anche le istruzioni **LAHF** e **SAHF** per la copia in **AH** del byte meno significativo del registro di flag o viceversa. Il flag di overflow si trova nel byte più significativo del registro dei codici di condizione e perciò non è interessato da queste istruzioni. Questo tipo d'istruzioni e i flag sono usati per lo più per la retrocompatibilità con i processori 8080 e 8085.

#### C.4.3 Cicli e operazioni iterative su stringhe

Scorrendo la tabella incontriamo poi le istruzioni di ciclo. L'istruzione **LOOP** decrementa il registro **CX** e, se il risultato è positivo, salta all'etichetta specificata. Anche le istruzioni **LOOPZ**, **LOOPE**, **LOOPNZ** e **LOOPNE** confrontano **CX** con **Z** per decidere se interrompere l'esecuzione del ciclo.

La destinazione di tutte le istruzioni **LOOP** deve trovarsi entro 128 byte dalla posizione corrente del program counter, perché l'istruzione contiene solo un offset di 8 bit con segno. Non si può calcolare con esattezza il numero d'istruzioni che è possibile saltare con 128 byte perché possono avere lunghezze differenti. In genere il primo byte delle istruzioni specifica il loro tipo, e alcune istruzioni occupano un solo byte nel segmento di codice. Spesso il secondo byte serve a definire i registri e le loro modalità e, se l'istruzione contiene spiazzamenti o dati immediati, la sua lunghezza può arrivare fino a quattro o sei byte. La lunghezza media delle istruzioni è di circa 2,5 byte, perciò **LOOP** consente in genere di saltare non più di 50 istruzioni in avanti o all'indietro.

Esistono anche meccanismi iterativi speciali per il trattamento delle stringhe, resi disponibili dalle istruzioni **REP**, **REPZ** e **REPNZ**. Anche le cinque istruzioni successive nella Figura C.4 definiscono gli operandi implicitamente e usano le modalità di autoincremento o di autodecremento dei registri indice. Tutte queste istruzioni prevedono che il registro indice **SI** punti al **segmento dati**, mentre **DI** punta al **segmento extra** localizzato all'indirizzo specificato da **ES**. L'istruzione **MOVSB** può essere usata insieme a **REP** per trasferire un'intera stringa in una sola istruzione. La lunghezza della stringa è contenuta nel registro **CX**. Dal momento che **MOVSB** non modifica i flag, non è possibile usare **REPNZ** durante la copia per verificare se un certo byte è equivalente al codice ASCII del carattere zero (carattere di terminazione delle stringhe), ma si può ovviare a questo problema usando **REPNZ SCASB** per memorizzare in **CX** la posizione del byte zero e poi usare questo valore in **REP MOVSB** per la copia. Questo concetto verrà chiarito dall'esempio di codice sulle stringhe esposto nel Paragrafo C.8. Nell'uso di queste istruzioni bisognerebbe prestare particolare attenzione al registro di segmento **ES**, a meno che **ES** e **DS** non assumano lo stesso valore. L'interprete si avvale di un modello piccolo di memoria e dunque impone **ES = DS = SS**.

#### C.4.4 Istruzioni di salto e di chiamata

L'ultimo gruppo d'istruzioni contiene i salti condizionati e non, le chiamate di subroutine e quelle di ritorno. L'operazione più semplice è svolta dall'istruzione **JMP** che può contenere come destinazione del salto un'etichetta o un qualsiasi indirizzo effettivo. Esistono due tipi di salto: **corto** e **lungo** (*near e far jump*). Si ha un salto corto quando la destinazione si trova nel segmento di codice corrente, che non può cambiare durante l'esecuzione dell'operazione. In caso di salto lungo invece il salto modifica il contenuto del registro **CS**. Nella versione con etichetta, il nuovo valore del registro del segmento di codice è contenuto nell'istruzione appena dopo l'etichetta, mentre nella versione con indirizzo effettivo si effettua il fetch di un dato long dalla memoria e lo si scomponete in due: la parola meno significativa corrisponde all'etichetta della destinazione, quella più significativa contiene il nuovo valore del registro del segmento di codice.

Naturalmente questa distinzione non sorprende affatto: per spostarsi in uno spazio degli indirizzi di 20 bit è necessario prevedere qualche meccanismo per poter utilizzare più di 16 bit. Il procedimento di salto termina con l'attribuzione di nuovi valori a **CS** e a **PC**.

#### Salti condizionati

L'8088 ha 15 istruzioni di salto condizionato, alcune delle quali hanno più di un nome (per esempio **JGE**, *Jump Greater or Equal*, è la stessa istruzione di **JNL**, *Jump Not Less than*); sono tutte elencate nella Figura C.5. Queste istruzioni consentono salti di al massimo 128 byte a

Istruzione	Descrizione	Condizione del salto
JNA, JBE	Al di sotto o uguale	CF=1 or ZF=1
JNB, JAE, JNC	Non al di sotto	CF=0
JE, JZ	Zero, uguale	ZF=1
JNLE, JC	Maggiore di	SF=OF and ZF=0
JGE, INL	Maggiore o uguale	SF=OF
JO	Overflow	OF=1
JS	Segno negativo	SF=1
JCXZ	CX vale zero	CX=0
JB, INAE, IC	Al di sotto	CF=1
JNBE, JA	Al di sopra	CF=0&ZF=0
JNE, JNZ	Non zero, diverso	ZF=0
JL, JNGE	Minore di	SF=OF
JLE, JNG	Minore o uguale	SF=OF or ZF=1
INO	No overflow	OF=0
JNS	Non negativo	SF=0

Figura C.5 Salti condizionati.

partire dall'istruzione. Se la destinazione è più distante di questo valore bisogna ricorrere a un meccanismo di costruzione di salto: si usa il salto con condizione opposta e lo si fa seguire da un salto incondizionato alla destinazione desiderata. L'effetto della combinazione di queste due istruzioni è proprio il salto a lungo raggio che si voleva ottenere. Per esempio, invece di scrivere

**JB FARLABEL**

scriviamo

**JNA 1f**  
**JMP FARLABEL**

1:

Dunque, se non è possibile effettuare una **JB** (*Jump Below*) verso l'etichetta lontana **FARLABEL** si usa una **JNA** (*Jump Not Above*) che specifica un'etichetta vicina / e la si fa seguire da un salto incondizionato all'etichetta lontana **FARLABEL**. L'effetto è lo stesso, ma richiede un consumo di risorse in tempo e spazio leggermente maggiore. Questo meccanismo complica un po' il calcolo delle distanze dei salti. Se una distanza è di poco minore di 128, ma alcune delle istruzioni fraposte tra il salto e la destinazione sono a loro volta salti condizionati, non è possibile stabilire la natura del salto (corto o lungo) finché non sono stati valutati i salti che si trovano nel mezzo. Per sicurezza, in questi casi l'assemblatore adotta un atteggiamento prudente e alle volte costruisce un salto in due istruzioni anche se non è strettamente necessario. Viceversa predilige il salto condizionato diretto solo quando è assolutamente certo che la destinazione si trovi alla portata di un salto corto.

Molti salti condizionati dipendono dai flag di stato e sono preceduti da un'istruzione di test o di confronto. Le istruzioni **CMP** sottraggono la sorgente dall'operando destinazione, impostano i codici di condizione e scartano il risultato. Nessuno degli operandi risulta modificato da queste istruzioni. Se il risultato è zero, o se il suo bit di segno vale 1 (cioè è negativo), il flag corrispondente viene asserito. Se il risultato non può essere espresso con il numero di bit disponibili, viene assegnato il flag di overflow. Se si verifica un riporto al di là del bit più significativo, il flag di riporto si attiva. I salti condizionati possono dipendere da diverse combinazioni di questi bit.

Se gli operandi sono intesi con segno bisogna usare **JG** (*Jump Greater than*) o **JL** (*Jump Less than*), mentre se sono senza segno si usano **JA** (*Jump Above*) e **JB** (*Jump Below*).

#### C.4.5 Chiamate di subroutine

L'8088 dispone di un'istruzione per la chiamata di procedure che, nella terminologia del linguaggio assemblativo, si chiamano **subroutine**. Analogamente al caso delle istruzioni di salto, anche per questo tipo d'istruzioni si distinguono le **chiamate ravvicinate** e quelle a **distanza** (*near e far call*). Solo le prime sono state implementate nell'interprete. La destinazione può essere un'etichetta o può trovarsi presso un indirizzo effettivo. I parametri necessari alla subroutine devono essere impilati sullo stack in ordine inverso, come illustrato nella Figura C.6. Nella terminologia del linguaggio assemblativo i parametri si chiamano **argomenti**, ma i due termini sono intercambiabili. Dopo il push degli argomenti si esegue l'istruzione **CALL**, che comincia con l'impilare sullo stack il valore corrente del program counter per salvare l'indirizzo di ritorno (quello dal quale bisogna riprendere l'esecuzione della routine chiamante al termine della subroutine).

A questo punto viene caricato il nuovo program counter a partire dall'etichetta o dall'indirizzo effettivo. Se la chiamata è a distanza, il registro del segmento di codice (cs) viene impilato prima del program counter (pc) e si procede al caricamento di entrambi a partire dai dati immediati o dall'indirizzo effettivo. Questo caricamento conclude l'esecuzione dell'istruzione **CALL**.

L'istruzione di ritorno, **RET**, non fa altro che recuperare l'indirizzo di ritorno dalla cima dello stack e lo scrive nel program counter, così l'esecuzione del programma riprende immediatamente dall'istruzione successiva alla **CALL**. Alle volte l'istruzione **RET** contiene un



Figura C.6 Un esempio di stack durante l'esecuzione di una subroutine.

#### C.4.5 Chiamate di subroutine

L'8088 dispone di un'istruzione per la chiamata di procedure che, nella terminologia del linguaggio assemblativo, si chiamano subroutine. Analogamente al caso delle istruzioni di salto, anche per questo tipo d'istruzioni si distinguono le chiamate ravvicinate e quelle a distanza (*near e far call*). Solo le prime sono state implementate nell'interprete. La destinazione può essere un'etichetta o può trovarsi presso un indirizzo effettivo. I parametri necessari alla subroutine devono essere impilati sullo stack in ordine inverso, come illustrato nella Figura C.6. Nella terminologia del linguaggio assemblativo i parametri si chiamano **argomenti**, ma i due termini sono intercambiabili. Dopo il push degli argomenti si esegue l'istruzione **CALL**, che comincia con l'impilare sullo stack il valore corrente del program counter per salvare l'indirizzo di ritorno (quello dal quale bisogna riprendere l'esecuzione della routine chiamante al termine della subroutine).

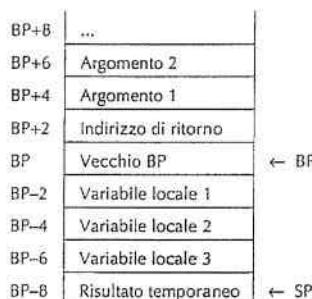


Figura C.6 Un esempio di stack durante l'esecuzione di una subroutine.

A questo punto viene caricato il nuovo program counter a partire dall'etichetta o dall'indirizzo effettivo. Se la chiamata è a distanza, il registro del segmento di codice (cs) viene impilato prima del program counter (pc) e si procede al caricamento di entrambi a partire dai dati immediati o dall'indirizzo effettivo. Questo caricamento conclude l'esecuzione dell'istruzione **CALL**.

L'istruzione di ritorno, **RET**, non fa altro che recuperare l'indirizzo di ritorno dalla cima dello stack e lo scrive nel program counter, così l'esecuzione del programma riprende immediatamente dall'istruzione successiva alla **CALL**. Alle volte l'istruzione **RET** contiene un numero positivo come dato immediato che rappresenta il numero di byte occupati dagli argomenti impilati sullo stack prima della chiamata; basta sommare questo numero a SP per rimuovere gli argomenti dallo stack. La variante a distanza **RETF** esegue la pop del registro del segmento di codice dopo la pop del program counter, proprio come ci si poteva aspettare.

Gli argomenti devono essere accessibili alla subroutine, perciò in genere la sua esecuzione comincia con il push del puntatore base e una copia del valore corrente di SP in BP. Così facendo, il puntatore base punta ora al suo valore precedente e l'indirizzo di

ritorno si trova alla posizione **BP + 2**, mentre il primo e il secondo argomento si trovano rispettivamente alla posizione **BP + 4** e **BP + 6**. Se la procedura ha delle variabili locali allora è possibile sottrarre il numero di byte richiesto dal valore del puntatore allo stack e le variabili possono essere indirizzate a partire dal puntatore base sommandogli offset negativi. Nell'esempio della Figura C.6 ci sono tre variabili locali di una parola ciascuna, localizzate agli indirizzi **BP - 2**, **BP - 4** e **BP - 6**. Dunque il registro **BP** utilizzabile per raggiungere l'intero insieme di argomenti e variabili locali della procedura corrente.

Lo stack viene usato nel modo consueto per contenere i risultati intermedi o per preparare gli argomenti per la chiamata successiva. Al ritorno da una subroutine lo stack può essere ripristinato senza calcolare lo spazio da essa utilizzato, ma basterà copiare il puntatore base nel puntatore allo stack, effettuare la pop del vecchio **BP** e infine eseguire l'istruzione **RET**.

Durante una chiamata di subroutine i valori di alcuni registri possono venir modificati. È buona pratica usare qualche tipo di convenzione, di modo che la routine chiamante non debba preoccuparsi dei registri usati dalla routine chiamata. La soluzione più semplice è adottare la stessa convenzione che esiste tra le chiamate di sistema e le subroutine ordinarie: la routine chiamata può modificare solo i registri **AX** e **DX**, perciò se uno di questi contiene informazioni importanti per il chiamante è consigliabile che questi ne salvi il contenuto nello stack prima d'impilare gli argomenti della chiamata. Se la subroutine usa anche altri registri allora può impilarli sullo stack immediatamente all'inizio della sua esecuzione e poi ripristinarli prima di chiamare l'istruzione **RET**. In breve, una buona convenzione richiede che il chiamante salvi **AX** e **DX** se li ritiene importanti, mentre il chiamato salva ogni altro registro che intende modificare.

#### C.4.6 Chiamate di sistema e subroutine di sistema

Per separare le operazioni su file (apertura, chiusura, lettura e scrittura) dalla programmazione in linguaggio assemblativo, i programmi sono eseguiti a livello superiore rispetto a quello del sistema operativo. L'interprete è stato dotato di sette chiamate di sistema e cinque funzioni (elencate nella Figura C.7) che ne permettono l'esecuzione su diverse piattaforme hardware.

Queste routine possono essere attivate con la sequenza di chiamata standard: si comincia con il push degli argomenti sullo stack in ordine inverso, si impila il numero di chiamata e infine si esegue l'istruzione **SYS** senza operandi per la trap di sistema. La routine di sistema reperisce nello stack tutte le informazioni che le sono necessarie, compreso il numero di chiamata e il servizio di sistema richiesto. I risultati sono restituiti nel registro **AX** o nella combinazione di registri **DX:AX** (se il risultato è di tipo long).

L'istruzione **SYS** non modifica i valori degli altri registri. Al suo completamento, gli argomenti restano impilati sullo stack e vanno rimossi (a cura del chiamante) agendo sul puntatore allo stack, a meno che non servano per una chiamata successiva.

Per comodità, è possibile definire all'inizio del programma assemblativo alcune costanti per i nomi delle chiamate di sistema, in modo da poterle richiamare per nome invece che per numero. Gli esempi che seguono ci permetteranno di descrivere in dettaglio molte chiamate di sistema, perciò in questo paragrafo ne diamo un resoconto abbreviato.

No.	Nome	Argomenti	Valore restituito	Descrizione
5	_OPEN	*name, 0/1/2	descrittore di file	Apre un file
8	CREAT	*name, *mode	descrittore di file	Crea un file
3	READ	fd, buf, nbytes	# di byte	Legge n byte nel buffer buf
4	WRITE	fd, buf, nbytes	# di byte	Scrive n byte nel buffer buf
6	CLOSE	fd	0 se ha successo	Chiude il file con descrittore fd
19	LSEEK	fd, offset(long), 0/1/2	posizione (long)	Sposta il puntatore del file
1	EXIT	status		Interrompe un processo
117	_GETCHAR		carattere letto	Legge un carattere da standard input
122	_PUTCHAR	char	byte scritto	Scrive un carattere sullo standard output
127	_PRINTF	*format, arg		Stampa formattata sullo standard output
121	_SPRINTF	buf, *format, arg		Stampa formattata nel buffer buf
125	_SSCANF	buf, *format, arg		Legge gli argomenti dal buffer buf

Figura C.7 Alcune chiamate di sistema UNIX e alcune subroutine implementate nell'interprete.

L'apertura dei file si effettua con le chiamate OPEN e CREAT, che prendono come primo argomento l'indirizzo della stringa contenente il nome del file. Il secondo argomento di OPEN vale 0 (se il file va aperto in lettura), 1 (se in scrittura) o 2 (lettura/scrittura). Se il file è scrivibile, ma non esiste, allora la chiamata provvede a crearlo. La chiamata CREAT produce un file vuoto con i permessi specificati dal secondo argomento. OPEN e CREAT restituiscono entrambe un piccolo intero nel registro AX; questo numero si chiama **descrittore di file** e può essere usato per lettura, scrittura o chiusura del file. Un descrittore di file negativo indica il fallimento della chiamata. All'avvio di un programma vengono aperti automaticamente tre file con descrittori 0 (standard input), 1 (standard output) e 2 (standard error output).

Le chiamate READ e WRITE hanno tre argomenti: il descrittore di file, un buffer per i dati e il numero di byte da trasferire. Dal momento che gli argomenti sono impilati in ordine inverso, si effettua prima il push del numero di byte, e poi i push dell'indirizzo del buffer, del descrittore di file e infine del numero di chiamata (READ o WRITE). Quest'ordine rispetta la sequenza degli argomenti della chiamata C standard

```
read(fd, buffer, bytes);
```

i cui parametri sono impilati nell'ordine *bytes, buffer e fd*.

La chiamata CLOSE richiede solo il descrittore di file e restituisce 0 in AX se la chiusura del file ha avuto successo. La chiamata EXIT richiede di impilare sullo stack lo stato di uscita del processo e non restituisce alcun valore.

La chiamata LSEEK modifica il **puntatore di lettura/scrittura** di un file già aperto. Il primo argomento è il descrittore di file, e il secondo è un dato long, perciò va scomposto nelle due parole che lo costituiscono e va impilata per prima la metà più significativa, anche nel caso in cui l'offset fosse contenuto tutto nella metà meno significativa. Il terzo argomento indica se il puntatore di lettura/scrittura va calcolato relativamente all'inizio del file (caso 0), alla posizione corrente (caso 1) o alla fine del file (caso 2). Il valore restituito è la nuova posizione del puntatore relativa all'inizio del file, memoriz-

zato come long. La funzione GETCHAR legge un carattere dall'input standard, lo salva in AL e azzerà AH. In caso di fallimento AX è posto invece a -1. La chiamata PUTCHAR scrive un byte sullo standard output e, in caso di successo, restituisce lo stesso byte scritto, mentre restituisce -1 in caso di fallimento.

La chiamata PRINTF stampa informazioni formattate. Il primo argomento della chiamata è l'indirizzo della stringa di formattazione, che specifica il formato dell'output. La stringa "%d" indica che l'argomento successivo è un intero presente nello stack e che quindi va convertito in notazione decimale prima della stampa. Allo stesso modo, "%x" indica il formato esadecimale e "%o" richiede la conversione in ottale. Inoltre, "%s" indica che l'argomento successivo è una stringa che termina con il carattere zero; lo stack contiene l'indirizzo di memoria dove reperirla. Il numero di argomenti che si trova sullo stack dovrebbe corrispondere al numero d'indicazioni di conversioni presente nella stringa di formattazione.

Per esempio, la chiamata C

```
printf("x = %d e y = %d\n", x, y);
```

stampa la stringa in cui le due occorrenze di "%d" sono state sostituite con i valori numerici di x e y. Anche in questo caso, per ragioni di compatibilità con il C, l'ordine dei push è "y", "x" e infine l'indirizzo della stringa di formattazione. Il motivo di questa convenzione è che la *printf* ha un numero variabile di parametri e quindi impilandoli in ordine inverso si è certi di poter individuare facilmente la stringa, che occupa sempre l'ultimo posto. Se i parametri fossero stati impilati da sinistra a destra la stringa di formattazione si troverebbe in qualche locazione precedente dello stack e la procedura *printf* non saprebbe come rintracciarla.

Per quanto riguarda la SPRINTF, il suo primo argomento è il buffer che riceve la stringa da stampare al posto dello standard output. Gli altri argomenti sono gli stessi di PRINTF. La chiamata SSCANF è il contrario di PRINTF nel senso che il suo primo argomento è una stringa contenente gli interi in notazione decimale, ottale o esadecimale, mentre l'argomento successivo è la stringa di formattazione contenente le indicazioni per le conversioni. Gli altri argomenti sono gli indirizzi delle parole di memoria che devono accogliere le informazioni dopo la conversione. Queste subroutine di sistema sono molto versatili e il Paragrafo C.8 ne presenta vari esempi di utilizzo.

#### C.4.7 Osservazioni sull'insieme d'istruzioni

La definizione ufficiale dell'8088 comprende un prefisso di deroga di segmento (*segment override*) che facilita la possibilità di utilizzare indirizzi effettivi che provengono da segmenti differenti; il primo indirizzo di memoria indicato dopo il prefisso viene risolto usando il registro di segmento corrispondente al segmento indicato. Per esempio, l'istruzione

```
ESEG MOV DX, (BX)
```

calcola per prima cosa l'indirizzo di BX usando il segmento extra, poi trasferisce il suo contenuto in DX. Tuttavia non è possibile derogare all'uso del segmento di stack quando l'indirizzo specificato si basa su SP, né all'uso del segmento extra in caso d'istruzione

sulle stringhe che utilizza il registro DI. I registri di segmento SS, DS ed ES istruzioni MOV, ma non è permesso trasferire dati immediati nei registri di segmento né utilizzarli nell'operazione XCHG. La programmazione con le deroghe di segmento e con le modifiche dei segmenti di registro è abbastanza complicata e dovrebbe essere evitata laddove possibile. L'interprete usa segmenti di registro costanti e quindi non incappa in problemi del genere.

La maggior parte dei calcolatori mette a disposizione istruzioni in virgola mobile, disponibili direttamente nel processore, in un coprocessore separato, oppure solo attraverso l'interpretazione via software per mezzo di una specie di trap in virgola mobile. La trattazione di queste funzionalità esula dagli scopi di questa appendice.

## C.5 Assemblatore

Conclusa la trattazione dell'architettura dell'8088, ci occupiamo della sua programmazione in linguaggio assemblativo e in particolare degli strumenti che mettiamo a disposizione per il suo apprendimento. Affrontiamo per primo l'assemblatore, poi il tracer e infine forniamo qualche informazione pratica circa il loro utilizzo.

### C.5.1 Introduzione

Fin qui abbiamo fatto riferimento alle istruzioni mediante i loro nomi mnemonici, cioè quelle abbreviazioni facili da ricordare come ADD e CMP. Lo stesso abbiamo fatto con i registri, indicati per mezzo di nomi simbolici quali AX e BP. Un programma scritto usando nomi simbolici per le istruzioni e per i registri si chiama **programma in linguaggio assemblativo**. L'esecuzione di un tale programma richiede in primo luogo la sua traduzione in stringhe binarie che avviene a cura di un programma, l'**assemblatore** (*assembler*). Il risultato del processo di conversione (l'output dell'assemblatore) si chiama **file oggetto**. Molti programmi effettuano chiamate a subroutine assemblate precedentemente e riposte in librerie. L'esecuzione di tali programmi richiede la combinazione del file oggetto appena assemblato e delle subroutine di libreria (a loro volta file oggetto) in un solo **file binario eseguibile** a opera del **linker** ("redattore di collegamenti"). La traduzione si può dire ultimata solo quando il linker ha terminato di costruire il file binario eseguibile a partire da uno o più file oggetto. A questo punto il sistema operativo può leggere il file binario eseguibile caricato in memoria e gestirne l'esecuzione.

Il primo compito dell'assemblatore è redigere una **tabella dei simboli**, che associa i nomi di variabili e costanti simboliche e le etichette ai numeri binari che rappresentano. Le costanti definite direttamente nel programma possono essere aggiunte alla tabella dei simboli senza ulteriori elaborazioni. Questo lavoro viene svolto nella prima passata.

D'altro canto, le etichette rappresentano indirizzi il cui valore non è immediatamente deducibile e, per calcolarlo, l'assemblatore scandisce il programma riga per riga in quella che è chiamata la **prima passata**. Durante questa fase, l'assemblatore aggiorna un **contatore di locazioni** indicato con il simbolo "." che si pronuncia **dot** ("punto"). Ogni volta che si incontra un'istruzione o un'allocazione di memoria, il contatore di locazioni viene incrementato della dimensione di memoria necessaria a contenere l'e-

mento incontrato. Perciò, se le prime due istruzioni hanno dimensioni di 2 e 3 byte, un'etichetta che si antepone alla terza istruzione varrà 5.

Per esempio, se un programma inizia con le istruzioni seguenti

```
MOV AX, 6
MOV BX, 500
L:
```

allora il valore di L sarà 5.

All'inizio della **seconda passata** è noto il valore numerico di ogni simbolo. Dato che i nomi mnemonici delle istruzioni hanno valori numerici costanti, è ora possibile cominciare la **generazione del codice**. Le istruzioni sono lette di nuovo una per volta e il loro equivalente binario viene scritto nel file oggetto. Una volta assemblata l'ultima istruzione del programma, il file oggetto è completo.

### C.5.2 as88, un assemblatore basato su ACK

Questo paragrafo descrive i dettagli dell'assemblatore/linker **as88**, disponibile nel CD-ROM allegato e presso il nostro sito web. **as88** è in grado di funzionare in associazione con il tracer già presentato, fa parte dell'Amsterdam Compiler Kit (ACK) ed è stato scritto sul modello degli assemblatori UNIX piuttosto che su quello degli assemblatori per MS-DOS o Windows. Questo assemblatore usa il punto esclamativo (!) per i commenti: il contenuto di una parte di riga che segue un punto esclamativo costituisce un commento e non ha alcun effetto sulla produzione del file oggetto. Eventuali righe vuote sono trattate alla stregua dei commenti.

Per la memorizzazione del codice e dei dati dopo la traduzione l'assemblatore usa tre sezioni diverse. Le sezioni sono in relazione con i segmenti di memoria della macchina. La prima è la **sezione di TESTO** per le istruzioni del processore; il segmento dati ospita la **sezione DATI** per l'inizializzazione di memoria necessaria all'avvio del processo; l'ultima sezione si chiama **BSS (Block Started by Symbol)**, che appartiene ancora al segmento dati ed è destinata all'allocazione della memoria non inizializzata (cioè inizializzata a 0). Ognuna di queste sezioni ha il proprio contatore di locazioni. Lo scopo delle sezioni è permettere all'assemblatore di generare porzioni d'istruzioni e porzioni di dati in modo indipendente, lasciando al linker il compito di riorganizzarle in modo che le istruzioni si trovino tutte nel segmento di testo e i dati nel segmento dati. Ogni riga del codice assemblativo viene tradotta in una sola sezione, ma le righe di codice e le righe di dati possono trovarsi frammiste le une alle altre. Durante l'esecuzione, la sezione di TESTO si trova allocata nel segmento di testo e le sezioni DATI e BSS sono memorizzate (consecutivamente) nel segmento dati.

Le istruzioni o le parole di dati dei programmi in linguaggio assemblativo possono essere precedute da un'etichetta, che può anche occupare un'intera riga (nel qual caso si intende faccia parte dell'istruzione o della parola dati della riga successiva). Per esempio in

```
CMP AX,ABC
JE L
MOV AX,XYZ
L:
```

l'etichetta *L* si riferisce all'istruzione o alla parola di dati che segue. Ci sono due tipi di etichette. Quelle globali sono identificatori alfanumerici seguite dai due punti (:), e quelle locali possono trovarsi solo nella sezione di TESTO e sono costituite da un'unica cifra seguita dal carattere due punti (:). Le etichette globali devono essere uniche e non possono corrispondere ad alcuna parola chiave del linguaggio, né a un nome mnemonico di un'istruzione. Le etichette locali possono invece occorrere più volte. L'istruzione

`JE 2f`

specificava che il salto JE (*Jump Equal*) è in avanti (*f* sta per *forward*) verso la successiva etichetta 2. Analogamente

`JNE 4b`

specificava che il salto JNE (*Jump Not Equal*) è all'indietro (*b* sta per *backward*) verso la precedente etichetta 4 più vicina.

L'assemblatore permette l'attribuzione di nomi simbolici alle costanti mediante la sintassi

`identificatore = espressione`

dove l'identificatore è una stringa alfanumerica come in

`BLOCKSIZE = 1024`

Tutti gli identificatori del linguaggio sono definiti dai loro primi otto caratteri: *BLOCKSIZE* e *BLOCKSIZZ* sono lo stesso simbolo, cioè *BLOCKSIZ*. Le espressioni si costruiscono a partire dalle costanti, dai valori numerici e dagli operatori. Le etichette sono considerate costanti perché il loro valore numerico è stabilito al termine della prima passata.

I valori numerici possono essere ottali (cominciano per 0), decimali o esadecimali (cominciano per 0x o 0X). I numeri esadecimali usano le lettere dalla "a" alla "f" (o dalla "A" alla "F") per rappresentare i valori da 10 a 15. Gli operatori interi sono +, -, \*, / e %, rispettivamente per addizione, sottrazione, moltiplicazione, divisione e resto. Gli operatori logici sono &, ^ e ~ per le operazioni bit a bit di AND, OR e NOT. Le espressioni possono contenere parentesi quadre, ma non tonde per non creare confusione con le modalità d'indirizzamento.

L'uso delle etichette nelle espressioni richiede cautela. Le etichette d'istruzioni non possono essere sottratte dalle etichette di dati. La differenza tra etichette confrontabili è un valore numerico, ma né le etichette né le loro differenze possono essere usate come costanti all'interno d'espressioni moltiplicative o logiche. Le espressioni consentite nelle definizioni di costanti possono essere utilizzate anche come costanti nelle istruzioni del processore. Alcuni assemblatori mettono a disposizione il costrutto delle macro-istruzioni che raggruppano sotto un unico nome un insieme d'istruzioni, ma *as88* non dispone di questa caratteristica.

Ogni linguaggio assemblativo definisce delle direttive che influiscono sul processo di assemblaggio, ma che non sono tradotte in codice binario e si dicono perciò pseudo-istruzioni. La Figura C.8 elenca le pseudoistruzioni di *as88*.

Istruzione	Descrizione
.SECT .TEXT	Assembra le linee seguenti nella sezione di TESTO
.SECT .DATA	Assembra le linee seguenti nella sezione DATI
.SECT .BSS	Assembra le linee seguenti nella sezione BSS
.BYTE	Assembra gli argomenti come una sequenza di byte
.WORD	Assembra gli argomenti come una sequenza di parole
.LONG	Assembra gli argomenti come una sequenza di long
.ASCII "str"	Salva la stringa str in ASCII senza farla terminare per zero
.ASCIZ "str"	Salva la stringa str in ASCII facendola terminare per zero
.SPACE n	Incrementa il contatore di locazioni di n posizioni
.ALIGN n	Allinea il contatore di locazioni alla locazione che dista al più n byte
.EXTERN	L'identificatore è un nome esterno

Figura C.8 Le pseudoistruzioni di *as88*.

Il primo gruppo specifica all'assemblatore la sezione in cui vanno elaborate le istruzioni successive. In genere queste indicazioni si scrivono su di una riga separata e possono essere inserite in qualsiasi punto del codice. Per ragioni di carattere implementativo bisogna usare per prima la sezione di TESTO, poi la sezione di DATI e infine la sezione BSS. Una volta specificati questi riferimenti iniziali, le sezioni possono essere usate in un ordine qualsiasi. Inoltre, la prima riga di una sezione dovrebbe cominciare sempre con un'etichetta globale. Non ci sono altri vincoli sull'ordine d'uso delle sezioni.

Il secondo gruppo di pseudoistruzioni contiene le dichiarazioni dei tipi di dati per il segmento dati: .BYTE, .WORD, .LONG e stringa. La parola chiave dei primi tre tipi può essere preceduta da un'etichetta facoltativa, seguita poi da una lista d'espressioni costanti separate da virgolette. Le stringhe mettono a disposizione due parole chiave, ASCII e ASCIZ, la cui unica differenza è che la seconda aggiunge in fondo alla stringa un byte di valore zero. Entrambe sono seguite da una stringa tra virgolette. La definizione delle stringhe consente l'uso di molte sequenze di escape tra cui menzioniamo quelle elencate nella Figura C.9. Oltre a ciò, è possibile inserire qualsiasi carattere con un carattere barra seguito dalla sua rappresentazione ottale, come \377 (si usano al massimo tre cifre e in questo caso non c'è bisogno di anteporre lo 0).

La pseudoistruzione SPACE causa l'incremento del puntatore alle locazioni del numero di byte specificato come argomento. Viene usata spesso dopo un'etichetta del segmento BSS per riservare lo spazio per l'allocazione di una variabile. La parola chiave ALIGN serve a far avanzare il puntatore in corrispondenza della prima locazione di memoria che dista 2, 4 o 8 byte dalla locazione corrente per facilitare l'assemblaggio di parole, long, e altro, allocando una quantità di memoria adeguata. Infine, la parola chiave EXTERN serve a dichiarare routine o locazioni di memoria come esterne, perché il linker le possa utilizzare per riferimenti esterni. Non è necessario che la loro definizione si trovi nel file corrente; può trovarsi ovunque, basta che sia raggiungibile dal linker.

Sequenza di escape	Descrizione
\n	Invio a capo (line feed)
\t	Tab
\W	Backslash
\b	Backspace
\f	Avanzamento modulo (form feed)
\r	Ritorno a margine (carriage return)
\v	Virgolette

Figura C.9 Alcune sequenze di escape di *as88*.

Benché l'assemblatore sia di per sé uno strumento abbastanza generale, bisogna segnalare alcuni aspetti da tener in conto quando lo si usa con il tracer. L'assemblatore riconosce le parole chiave scritte sia in minuscolo sia in maiuscolo, ma il tracer le visualizza sempre in maiuscolo. L'assemblatore accetta i caratteri “\r” e “\n” come caratteri di invio a capo (fine riga), ma il tracer usa sempre il secondo. Inoltre, anche se l'assemblatore può gestire programmi suddivisi in più file, il tracer richiede invece che il programma sia contenuto in un unico file con estensione “.S”. Al suo interno è possibile specificare direttive per includere altri file con il comando

```
#include filename
```

Così facendo il file richiesto viene ricopiato all'interno del file “.S” a partire dalla posizione del comando d'inclusione. L'assemblatore verifica che il file da includere sia già stato elaborato e ne carica una copia, il che è particolarmente utile nei casi in cui diversi file usino uno stesso file d'intestazione. Il comando `#include` deve trovarsi all'inizio di una riga, senza essere preceduto da spazi, e il percorso del file deve essere scritto tra virgolette.

Se c'è un unico file sorgente, *pr.s*, allora il progetto si chiama *pr*, e il file che combina i diversi sorgenti (in questo caso uno solo) sarà *pr.S*. Se c'è più di un file sorgente, allora il nome del progetto è il nome del primo file e funge anche da nome del file “.S”, generato dall'assemblatore concatenando il contenuto dei file sorgenti. Questo comportamento può essere modificato da linea di comando indicando l'opzione “-o nomeprogetto” prima del primo file sorgente, nel qual caso il file dei sorgenti combinati si chiamerà *nomeprogetto.S*.

L'inclusione dei file e la presenza di più file sorgenti comporta anche alcuni svantaggi. Innanzitutto è necessario che non ci siano nomi di etichette, di variabili e di costanti presenti in file diversi. Inoltre, dato che il file elaborato dall'assemblatore è in definitiva *nome-progetto.S*, tutti gli eventuali avvisi (*warning*) e messaggi d'errore si riferiscono alle sue righe di codice e non ai singoli file sorgenti. Per progetti molto piccoli, è spesso preferibile inserire tutto il programma in un unico file ed evitare di usare `#include`.

### C.5.3 Differenze tra assemblatori dell'8088

L'assemblatore *as88* è modellato sugli assemblatori standard di UNIX e, come tale, differisce dagli altri assemblatori per l'8088: MASM (di Microsoft) e TASM (di Borland), progettati per MS-DOS (e alcuni aspetti di ogni assemblatore sono intimamente legati al sistema operativo). MASM e TASM supportano entrambi tutti i modelli di memoria dell'8088 consentiti in MS-DOS. Per esempio supportano il modello minuscolo (*tiny*) di memoria, secondo cui tutto il codice e i dati devono essere contenuti in 64 KB, il modello **piccolo** (*small*), per cui il segmento di codice e il segmento dati possono raggiungere ciascuno i 64 KB, e i modelli **grandi** (*large*), che ammettono molteplici segmenti per il codice e per i dati. La differenza tra questi modelli dipende dall'uso che si fa dei registri di segmento. Il modello grande consente le chiamate a distanza e le modifiche del registro DS. Lo stesso processore introduce alcune limitazioni sui registri di segmento (per esempio il registro CS non può essere la destinazione di un'istruzione MOV). Per semplificare l'attività del tracer, *as88* usa un modello di memoria che somiglia al modello piccolo, anche se, quando è usato senza il tracer, può gestire i registri di segmento senza ulteriori restrizioni.

Questi altri assemblatori non hanno una sezione .BSS e inizializzano la memoria solo all'interno delle sezioni DATI. Il file in ingresso all'assemblatore comincia in genere con delle informazioni d'intestazione seguite dalla sezione DATI, indicata con la parola chiave .data, seguita a sua volta dal testo del programma dopo la parola chiave .code. L'intestazione usa la parola chiave title per specificare il nome del programma, la parola chiave .model per indicare il modello di memoria e la parola chiave .stack per allocare la memoria del segmento di stack. Se il file binario ha estensione .com, allora si usa il modello minuscolo di memoria, si considerano i registri come fossero tutti uguali e si riservano i primi 256 byte del segmento condiviso come “Prefisso del Segmento di Programma”.

Invece delle direttive .WORD, .BYTE e ASCIZ, questi assemblatori usano la parola chiave DW per le parole e DB per i byte. Dopo la direttiva DB è possibile definire una stringa racchiudendola tra virgolette. Le etichette per la dichiarazione di dati non sono seguite dai due punti. La parola chiave DUP serve a inizializzare grandi porzioni di memoria; è preceduta da un numero ed è seguita da un'inizializzazione, come nell'istruzione

```
LABEL DB 1000 DUP (0)
```

che inizializza i 1000 byte di memoria che seguono l'etichetta *LABEL* assegnando loro il byte ASCII zero.

Anche le etichette per le subroutine non sono seguite dai due punti, ma dalla parola chiave PROC. L'etichetta è ripetuta anche alla fine della subroutine ed è seguita dalla parola chiave ENDP, così l'assemblatore può dedurre l'estensione della subroutine. Non sono supportate le etichette locali.

Le parole chiave delle istruzioni sono identiche per i tre assemblatori MASM, TASM e *as88*. La sorgente segue sempre la destinazione in tutte le istruzioni a due operandi. È pratica comune l'utilizzo dei registri per il passaggio di argomenti alle funzioni invece di imparirli sullo stack. Tuttavia, se le routine assemblate vengono eseguite all'interno di programmi C o C++, è consigliabile usare lo stack per conformarsi al meccanismo di

chiamata delle subroutine C. Questa non è una reale differenza, perché anche in *as88* è possibile usare i registri invece dello stack per il passaggio degli argomenti.

La differenza maggiore tra questi tre assemblatori sta nelle chiamate di sistema. In MASM e in TASM, queste vengono effettuate tramite un interrupt di sistema INT. L'interrupt più comune è INT 21H che serve alla chiamata di funzioni MS-DOS. Il numero di chiamata è inserito in AX e ancora una volta si usa un registro per il passaggio degli argomenti. Esistono vettori di interrupt e numeri di interrupt diversi per i diversi dispositivi; per esempio INT 16H serve a chiamare le funzioni del BIOS per la tastiera, mentre INT 10H riguarda lo schermo. La programmazione di queste funzioni richiede al programmatore la conoscenza di un gran numero d'informazioni che variano da dispositivo a dispositivo. Le chiamate di sistema di UNIX messe a disposizione da *as88* sono invece molto più facili da usare.

## C.6 Tracer

Il tracer-debugger è stato pensato per l'esecuzione su un normale terminale con schermo a 24 × 80 caratteri (VT100) e comandi ANSI standard per i terminali. Se lo si vuole eseguire su macchine UNIX o Linux allora si può usare l'emulatore di terminale messo a disposizione dal sistema X-window. Sulle macchine Windows bisogna caricare il driver *ansi.sys* all'interno dei file d'inizializzazione del sistema nel modo che descriviamo in seguito. La Figura C.10 mostra la suddivisione della schermata del tracer in sette finestre.

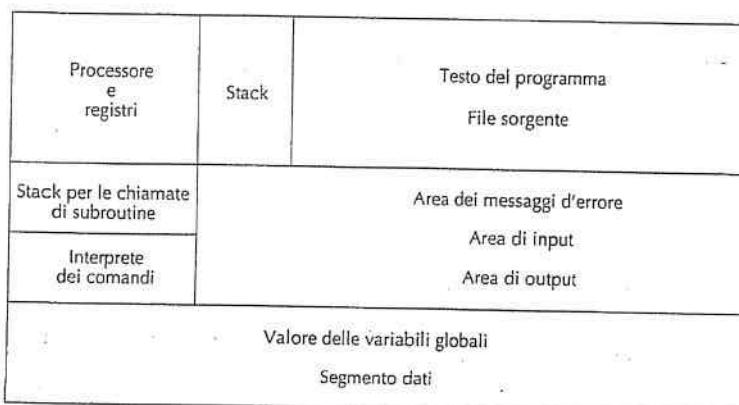


Figura C.10 Finestre del tracer.

La finestra in alto a sinistra è la finestra del processore; mostra il contenuto dei registri d'uso generale in notazione decimale, mentre gli altri sono in notazione esadecimale. Poiché il valore numerico del program counter non è di per sé molto informativo, la

posizione del sorgente di programma correntemente in esecuzione viene indicata in base alla precedente etichetta globale. Al di sopra del program counter sono elencati i valori di cinque codici di condizione: il simbolo "v" indica un evento di overflow (flag O), mentre le modalità d'incremento o di decremento sono indicate con i simboli ">" e "<" (flag D). Il flag di segno può valere "n" o "p" a seconda che il risultato dell'istruzione sia negativo oppure no. Il flag di zero vale "z" se asserito, mentre il bit di riporto vale "c" quando asserito. Il simbolo "-" indica un flag azzerato.

La finestra in alto al centro visualizza lo stack, rappresentato in esadecimale. La posizione del puntatore allo stack è indicata dalla freccia "=>". L'indirizzo di ritorno delle subroutine è indicato mediante una cifra anteposta al valore esadecimale. La finestra in alto a destra mostra una parte del file sorgente che contiene l'istruzione dell'esecuzione successiva. La freccia "=>" in questa finestra indica la posizione del program counter.

La finestra sotto quella del processore mostra le posizioni del codice contenenti le chiamate di subroutine effettuate più di recente. Appena sotto c'è la finestra dei comandi del tracer in cui viene visualizzata la storia dei comandi impartiti precedentemente, seguita dal cursore. Si noti che ogni comando deve essere seguito dalla pressione del tasto Invio.

La finestra in basso può contenere sei elementi della memoria dei dati globali. Ogni elemento comincia specificando una posizione relativa a un'etichetta, seguita dalla posizione assoluta nel segmento dati, da due punti e quindi da otto byte in esadecimale. Le 11 posizioni successive sono riservate a caratteri, seguiti dalla rappresentazione decimale di quattro parole. I byte, i caratteri e le parole rappresentano tutti lo stesso contenuto di memoria, anche se i caratteri contengono tre byte in più perché all'inizio non è noto se i dati saranno usati come interi con o senza segno, o come stringa.

La prima riga della finestra per l'input/output è riservata all'output per i messaggi d'errore del tracer, la seconda all'input e le righe successive all'output. L'output per gli errori è indicato con la lettera "E", l'input con la lettera "T" e lo standard output con il simbolo ">". La riga di input contiene una freccia "=>" a indicare il puntatore alla posizione di prossima lettura. Se il programma chiama *read* o *getchar*, l'input va immesso nell'area di input e bisogna farlo seguire dalla pressione del tasto Invio. La freccia "=>" indica la parte della riga che non è stata ancora elaborata.

In genere il tracer usa lo standard input per ricevere comandi e dati, ma è possibile anche preparare un file con i comandi per il tracer e uno con le righe di input che si vogliono far leggere prima che il controllo torni allo standard input. I file per i comandi al tracer hanno estensione .t, e quelli per l'input di dati hanno estensione .i. Il linguaggio assemblativo ammette l'uso indistinto di caratteri minuscoli o maiuscoli per la scrittura delle parole chiave, delle subroutine di sistema e delle pseudoistruzioni. Durante il processo di assemblaggio viene creato il file d'estensione .S e, al proprio interno, le parole chiave sono convertite in maiuscolo e vengono eliminati i caratteri di invio a capo. Dunque, per un dato progetto *pr* possono esistere sei file diversi:

1. *pr.s* per il codice del sorgente assemblativo;
2. *pr.\$* per il file dei sorgenti combinati;

3. *pr.88* per il file di caricamento;
4. *pr.i* per lo standard input preparato anticipatamente;
5. *pr.t* per i comandi all'interprete preparati anticipatamente;
6. *pr.#* per collegare il codice assemblativo al file di caricamento.

L'ultimo file viene usato dal tracer per riempire la finestra in alto a destra e visualizzare il program counter. Infine, il tracer controlla che il file di caricamento sia stato creato successivamente all'ultima modifica apportata al sorgente di programma, altrimenti emette un avviso al riguardo.

### C.6.1 Comandi del tracer

La Figura C.11 elenca i comandi del tracer. Il più importante è indicato nella prima riga ed è il comando che si immette con la semplice pressione del tasto Invio per richiedere l'esecuzione di una sola istruzione da parte del processore. Altrettanto importante è il comando *q* di uscita (*quit*), indicato all'ultima riga della tabella. Se si immette come comando un numero, questo provoca l'esecuzione di un numero corrispondente d'istruzioni: il numero *k* equivale a premere il tasto Invio *k* volte. Si ottiene lo stesso effetto se il numero è seguito da un punto esclamativo, *!*, o da una *X*.

Il comando *g* può essere usato per spostarsi a una certa riga del file sorgente. In particolare, se preceduto da un numero di riga, il tracer esegue tutte le istruzioni che la precedono; se si usa l'etichetta */T*, con o senza *+#*, il numero di riga dove fermarsi, si calcola a partire dall'etichetta d'istruzione *T*; l'immissione di *g* senza alcun'altra indicazione fa sì che il tracer esegua altri comandi finché non raggiunga di nuovo il numero di riga corrente.

Il comando */label* cambia significato a seconda che lo si usi per etichette di dati o d'istruzioni. Nel primo caso viene aggiunta o sostituita una riga della finestra più in basso con l'insieme di dati che si trova presso quell'etichetta. Nel secondo caso, il comando si comporta come il comando *g*. L'etichetta può essere seguita da un segno *+* o da un numero (indicato dal simbolo *#* nella Figura C.11) per specificare un offset relativo all'etichetta.

È possibile impostare un *breakpoint* ("punto d'interruzione") in corrispondenza di un'istruzione tramite il comando *b*, che può essere preceduto da un'etichetta d'istruzione ed eventualmente da un offset. Se durante l'esecuzione si raggiunge un'istruzione con breakpoint, il tracer si ferma; per farlo ripartire basta premere il tasto Invio o un comando d'esecuzione. Se non vengono specificati alcun numero e alcuna etichetta, il breakpoint è associato alla riga corrente. Per rimuovere un breakpoint si usa il comando *c*, preceduto eventualmente dagli stessi parametri opzionali di *b* (etichetta e numero). Esiste il comando *r* che ordina al tracer di continuare l'esecuzione finché non incontri un breakpoint, una chiamata di uscita o la fine del programma.

Il tracer tiene anche traccia del livello di subroutine in cui sta girando il programma. Si può desumere dalla finestra sotto la finestra del processore o anche dai numeri indicati nella finestra dello stack. Ci sono tre comandi che riguardano il livello di subroutine in esecuzione. Il comando *-* richiede al tracer di proseguire nell'esecuzione finché non raggiunga un livello di subroutine inferiore rispetto a quello attuale. L'effetto di questo

comando è l'esecuzione delle istruzioni che mancano al completamento della subroutine correntemente in esecuzione. Al contrario il comando *+* provoca il proseguimento dell'esecuzione fino al livello di subroutine appena superiore. Il comando *=* fa proseguire l'esecuzione finché non si incontra lo stesso livello di quello corrente e può essere usato per eseguire una sola subroutine quando ci si trova all'istruzione *CALL*. Quando si usa *=* la finestra del tracer non mostra i dettagli della subroutine. Si tratta di un comando particolarmente utile in presenza d'istruzioni *LOOP*; infatti l'esecuzione si ferma esattamente alla fine di ogni ciclo.

Indirizzo	Comando	Esempio	Descrizione
			Esegue una istruzione
#	, !, X	24	Esegue # istruzioni
/T+#	g, !,	/start+5g	Continua fino alla riga # dopo etichetta T
/T+#	b	/start+5b	Inserisce breakpoint alla riga # dopo etichetta T
/T+#	c	/start+5c	Rimuove breakpoint alla riga # dopo etichetta T
#	g	108g	Esegue il programma fino alla riga #
	g	g	Esegue il programma finché non torna alla riga corrente
	b	b	Inserisce breakpoint alla riga corrente
	c	c	Rimuove breakpoint dalla riga corrente
	n	n	Esegue programma fino alla riga successiva
	r	r	Esegue programma fino a un breakpoint o alla fine
	\&=	\&=	Esegue programma fino allo stesso livello di subroutine
	-	-	Esegue programma fino al livello di subroutine corrente meno 1
	+	+	Esegue programma fino al livello di subroutine corrente più 1
/D+#		/buf+6	Mostra il segmento dati alla posizione etichetta+#
/D+#	d, !	/buf+6d	Mostra il segmento dati alla posizione etichetta+#+
R, CTRL L	R	R	Aggiorna le finestre
q	q	q	Interrompe il tracing, restituisce il controllo all'interprete di comandi

Figura C.11 Comandi del tracer. Ogni comando deve essere seguito da un invio a capo (tasto Enter o Invio). Una casella vuota indica la necessità di digitare il solo carattere di a capo. I comandi che non devono specificare alcun indirizzo hanno la casella della colonna Indirizzo vuota. Il simbolo *#* indica un offset intero.

### C.7 Installazione

In questo paragrafo spieghiamo come usare gli strumenti software disponibili. Per prima cosa è necessario individuare il software corrispondente alla propria piattaforma hardware (qui mettiamo a disposizione versioni già compilate per Solaris, UNIX, Linux e

Windows). Il software è reperibile nel CD-ROM allegato. Le directory principali del CD-ROM sono *BigendNx*, *LtlendNx* e *MSWindos*, e ciascuna contiene una directory *assembler* con il materiale. Le tre cartelle radice sono destinate rispettivamente ai sistemi UNIX big endian (per esempio le workstation di Sun), ai sistemi UNIX little endian (per esempio Linux per PC) e ai sistemi Windows.

Dopo averla decompressa o copiata, la directory *assembler* dovrebbe contenere i seguenti file e sottodirectory: *READ\_ME*, *bin*, *as\_src*, *trce\_src*, *examples* e *exercise*. I sorgenti precompilati si trovano nella directory *bin*, ma sono presenti anche nella directory *examples* per comodità.

Per farsi un'idea del funzionamento del sistema, basta portarsi nella directory *examples* ed eseguire

```
t88 HlloWrld
```

Questo comando corrisponde al primo esempio del Paragrafo C.8.

La directory *as\_src* contiene il codice sorgente dell'assemblatore. I file sorgenti sono scritti in C e per ricompilarli si usa il comando *make*. Nella directory dei sorgenti è presente anche il file *Makefile* che serve per la compilazione sulle piattaforme POSIX compatibili. Sulle piattaforme Windows la compilazione è svolta dal file batch *make.bat*. Dopo la compilazione potrebbe essere necessario copiare i file eseguibili in un'altra directory, o modificare la variabile d'ambiente PATH, affinché l'assemblatore *t88* e il tracer *t88* siano visibili dalla directory contenenti i sorgenti in linguaggio assemblativo. In alternativa, invece di digitare solo *t88* è possibile specificare l'intero percorso dell'eseguibile.

Sui sistemi Windows 2000 e XP è necessario installare il driver di terminale *ansi.sys* inserendo la riga

```
device=%systemRoot%\System32\ansi.sys
```

nel file di configurazione *config.nt*, che si trova al percorso

```
\winnt\system32\config.nt (Windows 2000)
\windows\system32\config.nt (Windows XP)
```

Sui sistemi Windows 95/98/ME il driver va inserito nel file *config.sys*. Nei sistemi UNIX e Linux il driver è in genere già installato.

## C.8 Esempi

Nei paragrafi da C.2 a C.4 abbiamo presentato il processore 8088, la sua memoria e le sue istruzioni. Il Paragrafo C.5 ha introdotto il linguaggio assemblativo di *as88* che usiamo in questo testo. Nei Paragrafi C.6 e C.7 abbiamo studiato il tracer e descritto l'installazione degli strumenti software. In teoria queste informazioni dovrebbero già essere sufficienti per la scrittura e il debug di programmi assemblativi con gli strumenti forniti. Ciononostante, a molti lettori potrebbe giovare la presentazione dettagliata di alcuni esempi di programmi assemblativi e il loro debug attraverso il tracer. Perciò qui

presentiamo i programmi d'esempio che abbiamo inserito nella directory *examples* del materiale di supporto. Suggeriamo al lettore di assemblarli tutti e di visualizzare la loro esecuzione con il tracer.

### C.8.1 Esempio Hello World

Cominciamo con l'esempio *HlloWrld.s* della Figura C.12. Il programma è mostrato nella finestra di sinistra e i numeri di riga sono indicati dopo il punto esclamativo (!), il simbolo usato dall'assemblatore per i commenti. Le prime tre righe contengono le definizioni di costanti per associare la rappresentazione interna di due chiamate di sistema e di un file di output ai loro nomi convenzionali.

<pre>_EXIT = 1          !1 _WRITE = 4         !2 _STDOUT = 1        !3 .SECT .TEXT        !4 start:             !5     MOV CX,de-hw   !6     PUSH CX         !7     PUSH HW         !8     PUSH _STDOUT    !9     PUSH _WRITE     !10     SYS             !11     ADD SP, 8       !12     SUB CX,AX       !13     PUSH CX         !14     PUSH _EXIT      !15     SYS             !16 .SECT .DATA        !17 hw:                !18 .ASCII "Hello World\n" !19 de: .BYTE 0        !20</pre>	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 2px;">CS: 00 DS=SS=ES: 002</td><td style="padding: 2px;">AH:00 AL:0c AX: 12</td><td style="padding: 2px;">BH:00 BL:00 BX: 0</td><td style="padding: 2px;">CH:00 CL:0c CX: 12</td><td style="padding: 2px;">DH:00 DL:00 DX: 0</td><td style="padding: 2px;">SP: 7fd8 SF O D S Z C =&gt;0004</td><td style="padding: 2px;">BP: 0000 CC - &gt; p - - 0001 =&gt;</td><td style="padding: 2px;">SI: 0000 IP:000c:PC 0000</td><td style="padding: 2px;">DI: 0000 start + 7 000c</td><td style="padding: 2px;">MOV CX,de-hw ! 6</td><td style="padding: 2px;">PUSH CX ! 7</td><td style="padding: 2px;">PUSH HW ! 8</td><td style="padding: 2px;">PUSH _STDOUT ! 9</td><td style="padding: 2px;">PUSH _WRITE !10</td><td style="padding: 2px;">SYS !11</td><td style="padding: 2px;">ADD SP,8 !12</td><td style="padding: 2px;">SUB CX,AX !13</td><td style="padding: 2px;">PUSH CX !14</td></tr> </table>	CS: 00 DS=SS=ES: 002	AH:00 AL:0c AX: 12	BH:00 BL:00 BX: 0	CH:00 CL:0c CX: 12	DH:00 DL:00 DX: 0	SP: 7fd8 SF O D S Z C =>0004	BP: 0000 CC - > p - - 0001 =>	SI: 0000 IP:000c:PC 0000	DI: 0000 start + 7 000c	MOV CX,de-hw ! 6	PUSH CX ! 7	PUSH HW ! 8	PUSH _STDOUT ! 9	PUSH _WRITE !10	SYS !11	ADD SP,8 !12	SUB CX,AX !13	PUSH CX !14
CS: 00 DS=SS=ES: 002	AH:00 AL:0c AX: 12	BH:00 BL:00 BX: 0	CH:00 CL:0c CX: 12	DH:00 DL:00 DX: 0	SP: 7fd8 SF O D S Z C =>0004	BP: 0000 CC - > p - - 0001 =>	SI: 0000 IP:000c:PC 0000	DI: 0000 start + 7 000c	MOV CX,de-hw ! 6	PUSH CX ! 7	PUSH HW ! 8	PUSH _STDOUT ! 9	PUSH _WRITE !10	SYS !11	ADD SP,8 !12	SUB CX,AX !13	PUSH CX !14		
*																			
hw																			

(a)

(b)

Figura C.12 (a) Programma *HlloWrld.s*. (b) Finestre del tracer durante la sua esecuzione.

La pseudoistruzione *.SECT* della riga 4 specifica che le righe successive fanno parte della sezione di TESTO, ovvero che si tratta d'istruzioni del processore. Analogamente, la riga 17 indica che le righe sottostanti costituiscono dati. La riga 19 inizializza una stringa di dati di 12 byte, compresi lo spazio e il carattere di fine riga (*\n*).

Le righe 5, 18 e 20 contengono etichette, individuabili dalla presenza dei due punti. Queste etichette rappresentano valori numerici e in questo sono simili a costanti; tuttavia, il valore numerico non è dato, ma deve essere calcolato dall'assemblatore. L'etichetta *start* si trova all'inizio della sezione di TESTO e quindi vale 0, ma il valore di un'eventuale etichetta successiva nella sezione di TESTO (nell'esempio non ce ne sono) dipenderebbe dal numero di byte che la precedono. Si consideri ora la riga 6 che termina con la differenza di due etichette, cioè con un valore costante. In effetti la riga è equivalente all'istruzione

```
MOV CX,12
```

con la differenza che, nel primo caso, il calcolo della lunghezza della stringa è lasciato all'assemblatore e non al programmatore. Il valore indicato dalla differenza corrisponde alla quantità di spazio riservato alla stringa della riga 19 nella sezione dati. L'istruzione MOV della riga 6 richiede la copia di *de -hw* in CX.

Le righe da 7 a 11 mostrano il funzionamento dell'invocazione di una chiamata di sistema. Le cinque righe sono la traduzione in codice assemblativo della chiamata di funzione C

```
write(1, hw, 12);
```

il cui primo parametro è il descrittore di file per lo standard output (1), il secondo è l'indirizzo della stringa da stampare (*hw*) e il terzo è la lunghezza della stringa (12). Le righe dalla 7 alla 9 impilano questi parametri sullo stack in ordine inverso, secondo la convenzione del C e quella adottata dal tracer. La riga 10 impila sullo stack il numero della chiamata di sistema write (4) e la 11 effettua la chiamata vera e propria. Questa sequenza di chiamata ricalca il funzionamento dei programmi in linguaggio assemblativo scritti per i sistemi UNIX (o Linux), ma per eseguirla direttamente su altri sistemi operativi dovrebbe essere leggermente modificata per aderire alle convenzioni di quei sistemi. Tuttavia, l'assemblatore *as88* e il tracer *t88* usano le convenzioni di chiamata di UNIX anche se sono eseguiti su sistemi Windows.

La chiamata di sistema della riga 11 esegue la scrittura vera e propria. La riga 12 ripristina lo stack alla posizione precedente alla chiamata, spostando il puntatore allo stack indietro di quattro parole di 2 byte ciascuna. Se l'operazione di scrittura ha successo, il numero di byte scritti viene restituito nel registro AX. La riga 13 sottrae il risultato della chiamata di sistema della riga 11 al valore del registro CX, contenente la lunghezza della stringa originale, per stabilire se la chiamata ha avuto successo, ossia per stabilire se sono stati scritti tutti i byte. Di conseguenza, lo stato di uscita del programma sarà 0 soltanto in caso di successo. Le righe 14 e 15 preparano l'esecuzione della chiamata di sistema exit (riga 16) impilando sullo stack lo stato d'uscita e il codice della funzione EXIT.

Le istruzioni MOV e SUB usano il primo operando come destinazione e il secondo come sorgente. Questa è la convenzione adottata dal nostro assemblatore, altri potrebbero usare l'ordine inverso; non c'è alcuna ragione particolare per preferire un ordinamento a un altro.

Proviamo ad assemblare e a eseguire *HelloWorld.s*. Forniamo le indicazioni sui procedimenti da seguire sulle piattaforme UNIX e Windows; le indicazioni per UNIX dovrebbero valere anche per Linux, Solaris, MacOS X e per altre varianti di UNIX. Si comincia con l'aprire una finestra dell'interprete di comandi (la shell). Su Windows bisogna cliccare in sequenza su

Start > Programmi > Accessori > Prompt dei comandi

Quindi ci si porta nella directory *examples* con il comando *cd* (*Change Directory*). L'argomento di questo comando dipende dalla locazione del sistema in cui è stato copiato il materiale di supporto. Una volta entrati nella directory si può verificare la presenza dei file binari dell'assemblatore e del tracer con il comando UNIX *ls* o con il comando

Windows *dir*. I file binari si chiamano rispettivamente *as88* e *t88*; sui sistemi Windows hanno estensione *.exe*, ma non c'è bisogno di specificare l'estensione per lanciare la loro esecuzione. Se i file suddetti non si trovano nella directory, bisogna trovarli e copiarli al suo interno.

È ora possibile passare all'assemblaggio digitando il comando

```
as88 HelloWorld.s
```

Se il comando produce un errore nonostante l'assemblatore sia presente nella cartella, si può provare a digitare

```
./as88 HelloWorld.s
```

nei sistemi UNIX o

```
.\as88 HelloWorld.s
```

nei sistemi Windows. Se l'assemblaggio termina correttamente, verranno visualizzati i messaggi:

```
Project HelloWorld listfile HelloWorld.S
Project HelloWorld num file HelloWorld.# 
Project HelloWorld loadfile HelloWorld.88.
```

che indicano la creazione dei file corrispondenti. Se non ci sono messaggi d'errore, si procede con l'invocazione del tracer:

```
t88 HelloWorld
```

che ne apre le finestre. La freccia nel riquadro in alto a destra punta ora sull'istruzione

```
MOV CX,de-hw
```

della riga 6. A questo punto si può digitare un "a capo" (il tasto Enter o Invio sulle tastiere dei PC) che provoca l'esecuzione di un'istruzione, per cui l'istruzione ora puntata è

```
PUSH CX
```

e CX contiene adesso il valore 12 (nella finestra di sinistra). Dopo un altro "a capo" la finestra centrale contiene il valore 000c, cioè 12 in esadecimale. Questa finestra mostra lo stack, che adesso contiene una parola di valore 12. Se si preme Invio ancora tre volte si eseguono le tre PUSH delle righe 8, 9 e 10, dopodiché lo stack conterrà quattro elementi e il program counter varrà 000b (indicato nella finestra di sinistra).

Dopo un altro "a capo" viene eseguita la chiamata di sistema che produce la stampa della stringa "Hello World\n" nella finestra in basso a destra. Adesso SP vale 0x7ff0, ma dopo un altro "a capo" viene incrementato di 8 e diventa 0x7ff8. Dopo altri quattro invii a capo, la chiamata di sistema viene completata e l'esecuzione del tracer si interrompe.

Per comprendere a fondo il funzionamento dell'assemblaggio può essere utile aprire il file *HelloWorld.s* con un editor. A tal fine conviene evitare di usare programmi come Word perché potrebbero formattare il sorgente in modo scorretto. Sulle macchine UNIX

si possono usare per esempio *ex*, *vi*, o *emacs*; sui sistemi Windows si può aprire *NotePad*, un editor molto semplice reperibile cliccando in sequenza su

Start > Programmi > Accessori > Notepad

Consigliamo di modificare la stringa alla riga 19 per visualizzare un messaggio diverso. Dopo aver salvato il file, si può procedere al suo assemblaggio ed eseguirlo con il tracer. Questa semplice modifica è un primo approccio alla programmazione in linguaggio assemblativo.

### C.8.2 Esempio con i registri d'uso generale

L'esempio che presentiamo ora mostra più in dettaglio l'uso dei registri ed evidenzia una delle insidie della moltiplicazione nell'8088. La parte sinistra della Figura C.13 mostra una porzione del programma *genReg.s*, mentre la parte destra raffigura la finestra dei registri del tracer in due momenti diversi dell'esecuzione del programma. La Figura C.13(b) mostra lo stato dei registri dopo l'esecuzione della riga 7. L'istruzione alla riga 4

MOV AX,258

carica il valore 258 in AX, che equivale a caricare 1 in AH e 2 in AL. Alla riga 5 si effettua la somma di AL e AH, dopodiché AH vale 3. Alla riga 6 viene copiato il contenuto della variabile *times* (10) in CX, alla riga 7 viene caricato l'indirizzo della variabile *muldat* (2) in BX; l'indirizzo di *muldat* è 2, perché questa si trova in corrispondenza del secondo byte della sezione DATI. Questa è la situazione fotografata nella Figura C.13(b): AH vale 3, AL vale 2 e AX vale 770, in quanto  $3 \times 256 + 2 = 770$ .

<pre> start:           13     MOV AX,258   14     ADDB AH,AL   15     MOV CX,(times) 16     MOV BX,muldat 17     MOV AX,(BX)   18     Ilp: MUL 2(BX) 19         LOOP Ilp 110     .SECT .DATA 111     times: .WORD 10 112     muldat:.WORD 625,2 113   </pre>	<pre> CS: 00 DS=SS=ES: 002 AH:03 AL:02 AX: 770 BH:00 BL:02 BX: 2 CH:00 CL:0a CX: 10 DH:00 DL:00 DX: 0 SP: 7fe0 SF O D S Z C BP: 0000 CC - &gt; p - SI: 0000 IP:0009:PC DI: 0000 start + 4   </pre>	<pre> CS: 00 DS=SS=ES: 002 AH:38 AL:80 AX: 14464 BH:00 BL:02 BX: 2 CH:00 CL:04 CX: 4 DH:00 DL:01 DX: 1 SP: 7fe0 SF O D S Z C BP: 0000 CC v &gt; p - c SI: 0000 IP:0011:PC DI: 0000 start + 7   </pre>
(a)	(b)	(c)

Figura C.13 (a) Porzione del programma. (b) Finestra dei registri del tracer dopo l'esecuzione della riga 7. (c) I registri.

L'istruzione successiva (riga 8) copia il contenuto di *muldat* in AX. Dunque, dopo un ulteriore invio a capo, AX conterrà 625.

Siamo ora pronti per entrare in un ciclo che moltiplica il contenuto di AX alla parola che si trova all'indirizzo 2(BX), cioè all'indirizzo *muldat* + 2, che vale 2. La destinazione implicita dell'istruzione di moltiplicazione è la coppia di registri DX:AX. Dopo la prima iterazione del ciclo il risultato può essere contenuto in una sola parola, perciò AX

contiene il risultato (1250) e DX resta a 0. La Figura C.13(c) mostra il contenuto dei registri dopo 7 moltiplicazioni successive.

All'inizio del ciclo AX valeva 625, perciò il risultato delle sette moltiplicazioni per 2 è 80.000. Questo numero non può più essere contenuto in AX, bensì occupa ora i 32 bit dei registri DX:AX accoppiati. DX vale 1 e AX vale 14.464. L'istruzione LOOP decrementa il registro CX di un'unità a ogni iterazione. Poiché all'inizio dell'esecuzione CX valeva 10, dopo sette operazioni MUL (e dopo sole sei iterazioni dell'istruzione LOOP) vale 4.

Il problema si verifica al momento della moltiplicazione successiva. La moltiplicazione coinvolge AX ma non DX, perciò MUL moltiplica AX (14.464) per 2 producendo 28.928. Il risultato è inserito in AX e DX viene azzerato, il che è algebricamente errato.

### C.8.3 Istruzioni di chiamata e puntatori ai registri

L'esempio successivo, *vecprod.s*, è un breve programma per il calcolo del prodotto scalare di due vettori, *vec1* e *vec2*. È mostrato nella Figura C.14.

La prima parte del programma prepara la chiamata di *vecmul* salvando SP in BP e impilando gli indirizzi di *vec2* e *vec1* sullo stack, di modo che *vecmul* possa accedervi. La riga 8 carica in CX la lunghezza in byte del vettore. La riga 9 fa scorrere questo valore di un bit a destra, così CX contiene ora la lunghezza del vettore in parole. Questo valore viene impilato sullo stack alla riga 10. La riga 11 effettua la chiamata a *vecmul*.

Ancora una volta vale la pena sottolineare come gli argomenti delle subroutine vengano impilati in ordine inverso per rispettare la convenzione vigente per le chiamate del linguaggio C. Così facendo, *vecmul* può essere invocata anche all'interno di un programma C con la chiamata

*vecmul(count, vec1, vec2)*

Durante l'esecuzione dell'istruzione CALL, l'indirizzo di ritorno viene impilato sullo stack e, usando il tracer, si evince che questo indirizzo è 0x0011.

La prima istruzione della subroutine è PUSH del puntatore base, BP, alla riga 22. BP viene salvato perché servirà a indirizzare gli argomenti e le variabili locali della subroutine. Successivamente viene copiato il puntatore allo stack (riga 23) di modo che il nuovo valore del puntatore base punti al suo vecchio valore.

A questo punto si è pronti per il caricamento nei registri degli argomenti e per l'allocatione dello spazio per le variabili locali. Le tre righe successive prelevano gli argomenti dallo stack e li salvano nei registri. Si ricordi che lo stack è orientato alla parola, perciò i suoi indirizzi dovrebbero essere sempre pari. L'indirizzo di ritorno si trova appena dopo il vecchio puntatore base, perciò viene indirizzato con 2(BP). Segue l'argomento *count*, indirizzato con 4(BP), che viene caricato in CX alla riga 24. Le righe 25 e 26 caricano *vec1* e *vec2* in SI e DI. Questa subroutine utilizza una variabile locale (inizializzata a 0) per salvare il risultato. A tal fine, la riga 27 esegue il push del valore 0.

La Figura C.15 mostra lo stato del processore appena prima del primo ingresso nel ciclo della riga 28. La finestra stretta che si trova in alto e al centro della figura (alla destra dei registri) mostra lo stack. Alla base dello stack c'è l'indirizzo di *vec2* (0x0022), seguito da quello di *vec1* (0x0018) e dal terzo argomento, cioè il numero d'elementi di ciascun vettore (0x0005).

```

_EXIT = 1
_PRINTF = 127
.SECT .TEXT
instart:
    MOV BP,SP
    PUSH vec2
    PUSH vec1
    MOV CX,vec2-vec1
    SHR CX,1
    PUSH CX
    CALL vecmul
    MOV (inprod),AX
    PUSH AX
    PUSH pfmt
    PUSH _PRINTF
    SYS
    ADD SP,12
    PUSH 0
    PUSH _EXIT
    SYS

    vecmul:
        PUSH BP
        MOV BP,SP
        MOV CX,4(BP)
        MOV SI,6(BP)
        MOV DI,8(BP)
        PUSH 0

        T: LODS
        MUL (DI)
        ADD -2(BP),AX
        ADD DI,2
        LOOP 1b
        POP AX
        POP BP
        RET

.SECT .DATA
pfmt: ASCIZ "Inner product is: %d\n"
.ALIGN 2
vec1:WORD 3,4,7,11,3
vec2:WORD 2,6,3,1,0
.SECT .BSS
inprod: .SPACE 2

```

! 1 definizione del valore di \_EXIT  
! 2 definizione del valore di \_PRINTF  
! 3 inizio del segmento di TESTO  
! 4 definizione dell'etichetta instart  
! 5 salvataggio di SP in BP  
! 6 push dell'indirizzo di vec2  
! 7 push dell'indirizzo di vec1  
! 8 CX = numero di byte nel vettore  
! 9 CX = numero di parole nel vettore  
! 10 push del numero di parole (count)  
! 11 chiamata di vecmul  
! 12 copia di AX  
! 13 push del risultato per la stampa  
! 14 push dell'indirizzo della stringa di formattazione  
! 15 push del codice di funzione di PRINTF  
! 16 chiamata della funzione PRINTF  
! 17 ripristino dello stack  
! 18 push del codice di stato  
! 19 push del codice di funzione di EXIT  
! 20 chiamata della funzione EXIT

! 21 inizio di vecmul(count, vec1, vec2)  
! 22 salvataggio di BP nello stack  
! 23 copia di SP in BP per l'accesso agli argomenti  
! 24 trasf. di count in CX per il controllo del ciclo  
! 25 SI = vec1  
! 26 DI = vec2  
! 27 push di 0 sullo stack

! 28 copia di (SI) in AX  
! 29 moltiplicazione di AX per (DI)  
! 30 somma di AX con valore accumulato in memoria  
! 31 incremento di DI per puntare a elemento successivo  
! 32 se CX > 0, vai all'etichetta 1b  
! 33 pop della cima dello stack in AX  
! 34 ripristino di BP  
! 35 ritorno da subroutine

! 36 inizio del segmento DATI  
! 37 definizione di stringa  
! 38 forza allineamento a indirizzo pari  
! 39 vettore 1  
! 40 vettore 2  
! 41 inizio del segmento BSS  
! 42 allocazione dello spazio per inprod

Figura C.14 Programma vecprods.s.

Poi c'è l'indirizzo di ritorno (0x0011); il numero 1 alla sua sinistra indica che si tratta di un indirizzo di ritorno di un livello inferiore rispetto al livello del programma principale. La stessa indicazione è presente anche nella finestra sotto i registri, ma questa volta l'indirizzo è indicato in maniera simbolica. Risalendo lo stack, sopra l'indirizzo di ritorno si trova il vecchio valore di BP (0x7fc0) e poi lo zero impilato alla riga 27. La freccia che punta a questo valore indica il valore puntato da SP. La finestra alla destra dello stack mostra un frammento del testo del programma, dove la freccia indica l'istruzione successiva da eseguire.

MOV BP,SP	! 5	CS: 00 DS=SS:ES: 004	PUSH BP	! 22
PUSH vec2	! 6	AH:00 AL:00 AX: 0	MOV BP,SP	! 23
PUSH vec1	! 7	BH:00 BL:00 BX: 0	MOV CX,4(BP)	! 24
MOV CX,vec2-vec1	! 8	CH:00 CL:05 CX: 5 =>0000	MOV SI,6(BP)	! 25
SHR CX,1	! 9	DH:00 DL:00 DX: 0 7fc0	MOV DI,8(BP)	! 26
PUSH CX	! 10	SP: 7fb4 SF O D S Z C 1 0011	PUSH 0	! 27
CALL vecmul	! 11	BP: 7fb6 CC - > p z - 0005 =>1:	LODS	! 28
vecmul :	! 21	SI: 0018 IP:0031:PC 0018	MUL (DI)	! 29
PUSH BP	! 22	DI: 0022 vecmul+7 0022	ADD -2(BP),AX	! 30
MOV BP,SP	! 23			
MOV CX,4(BP)	! 24			
MOV SI,6(BP)	! 25			
MOV DI,8(BP)	! 26			
PUSH 0	! 27			
1: LODS	! 28			
MUL (DI)	! 29			
ADD -2(BP),AX	! 30			
ADD DI,2	! 31			
LOOP 1b	! 32			

! 1 <= instart + 7

vec1 + 0 = 0018: 3 0 4 0 7 0 b 0 ..... 3  
vec2 + 0 = 0022: 2 0 6 0 3 0 1 0 ..... 2  
pfmt + 0 = 0000: 54 68 65 20 69 6e 20 70 The in' prod 26708  
pfmt + 18 = 0012: 25 64 21 a 0 0 3 0 %d! ..... 25537

Figura C.15 Esecuzione di vecprods.s al primo ingresso nel ciclo della riga 28.

Esaminiamo ora il ciclo che comincia alla riga 28. L'istruzione LODS carica in AX una parola di memoria dal segmento dati in modalità registro indiretto (tramite il registro SI). Poiché il flag della modalità d'incremento è asserito, LODS si autoincrementa, perciò dopo l'esecuzione della 28, SI punta al successivo elemento di vec1.

Per visualizzare graficamente questo meccanismo, basta avviare il tracer con il comando

t88 vecprod

e attendere il caricamento del programma, quindi digitare

/vecmul+7b

(seguito da un invio a capo) per inserire un breakpoint alla riga dell'istruzione LODS. Ricordiamo che ogni comando deve essere seguito da un invio a capo e d'ora in avanti eviteremo di ripeterlo. Digitare quindi il comando

g

affinché il tracer continui a eseguire le istruzioni finché non incontri il breakpoint; si fermerà alla riga contenente LODS.

La riga 29 moltiplica AX per l'operando sorgente. La parola di memoria richiesta dall'istruzione MUL è prelevata dal segmento dati attraverso il registro DI usato in modalità registro indiretto. La destinazione di MUL è la combinazione di registri DX:AX, che è implicita (cioè non menzionata dall'istruzione).

La riga 30 somma il risultato della moltiplicazione alla variabile locale che si trova all'indirizzo dello stack -2(BP). Dato che MUL non incrementa automaticamente il proprio operando, bisogna effettuare l'incremento esplicitamente (riga 31), dopodiché DI punterà all'elemento successivo del vettore vec2.

L'istruzione `LOOP` conclude questa fase di calcolo: `CX` viene decrementato e, se è ancora positivo, il programma torna all'etichetta locale `l` della riga 28. L'espressione `lb` sta a indicare proprio la precedente etichetta locale `l` più vicina rispetto alla posizione corrente. Alla terminazione del ciclo, la subroutine esegue la pop del valore da restituire in `AX` (riga 33), ripristina `BP` (riga 34) e passa il controllo al programma chiamante (riga 35).

Il programma riprende dalla posizione successiva alla chiamata con l'esecuzione dell'istruzione `MOV` alla riga 12. Questa è la prima di cinque istruzioni finalizzate alla stampa del risultato. La chiamata di sistema `printf` ricalca il comportamento della funzione `printf` della libreria C standard. I tre argomenti sono impilati sullo stack alle righe 13-15; si tratta del valore intero da stampare, dell'indirizzo della stringa di formattazione (`pfmt`) e del codice di funzione di `printf` (127). La stringa di formattazione `pfmt` contiene la stringa `%d` a indicare che la chiamata `printf` ha un argomento di tipo intero che deve essere inserito nella stampa.

La riga 17 ripristina lo stack. Poiché l'esecuzione del programma è cominciata alla riga 5 con il salvataggio del puntatore allo stack nella locazione del puntatore base, sarebbe stato possibile ripristinare lo stack anche con l'istruzione

```
MOV SP,BP
```

Il vantaggio di questa soluzione è che il programmatore non deve tenere il conto delle dimensioni raggiunte dal record d'attivazione. La questione è di poca importanza per quel che riguarda il programma principale, ma nelle subroutine può aiutare a liberare lo stack dai dati inutili, per esempio le variabili locali obsolete.

La subroutine `vecmul` può essere inclusa in altri programmi. Se il file sorgente `vecprod.s` è inserito nella riga dei comandi dopo un altro file sorgente in ingresso all'assemblatore, la subroutine è utilizzabile per moltiplicare due vettori di lunghezza fissa. Si consiglia in tal caso di rimuovere le definizioni delle costanti `_EXIT` e `_PRINTF` per evitare di definirle due volte. Basta includere in un punto qualsiasi il file `syscalnr.h` per non doversi più preoccupare di definire le costanti delle chiamate di sistema.

#### C.8.4 Debugging di un programma per la stampa di array

I programmi degli esempi precedenti sono molto semplici e svolgono la propria funzione correttamente. Veniamo ora all'uso del tracer come strumento di supporto per il debugging di programmi con errori. Il programma seguente dovrebbe stampare l'array d'interi che comincia all'etichetta `vec1`, tuttavia la sua versione iniziale contiene ben tre errori. Vedremo come usare l'assemblatore e il tracer per correggere questi errori; cominciamo con il descrivere il codice.

Per ragioni di comodità abbiamo creato il file d'intestazione `syscalnr.h` dove abbiamo inserito tutte le definizioni di costanti che identificano le chiamate di sistema, per non doverle ridefinire all'interno di ogni programma che ne ha bisogno. La riga 1 del programma serve proprio a includere il file d'intestazione. `..syscalnr.h` contiene anche le costanti dei descrittori di file

```
STDIN = 0
STDOUT = 1
STDERR = 2
```

aperti automaticamente all'avvio del processo, così come le etichette d'intestazione per i segmenti del testo e dei dati. Tutte queste definizioni sono molto utilizzate, perciò vale la pena includere `..syscalnr.h` all'inizio di tutti i file sorgente scritti in linguaggio assemblativo. Se il codice sorgente è distribuito tra più file, l'assemblatore provvede a includere il file d'intestazione una sola volta per evitare di ridefinire più volte le stesse costanti.

La Figura C.16 contiene il codice del programma `arrayprt`. Abbiamo omesso il commento delle singole istruzioni, perché dovrebbero risultare ormai familiari, perciò abbiamo utilizzato il formato a due colonne. La riga 4 inserisce l'indirizzo dello stack (ancora vuoto) nel registro puntatore base per consentire il successivo ripristino dello stack alla riga 10; basterà poi copiare il puntatore base nel puntatore allo stack come già descritto nell'esempio precedente. Le righe dalla 5 alla 9 impilano gli argomenti nello stack analogamente a quanto già visto negli esempi precedenti. Le righe dalla 22 alla 25 caricano i registri nella subroutine.

#include "..syscalnr.h"	! 1	.SECT .TEXT	! 20
vecprint:		PUSH BP	! 21
.SECT .TEXT	! 2	MOV BP,SP	! 22
vecstrt:	! 3	MOV CX,4(BP)	! 23
MOV BP,SP	! 4	MOV BX,6(BP)	! 24
PUSH vec1	! 5	MOV SI,0	! 25
MOV CX,frmstr-vec1	! 6	PUSH frmatkop	! 26
SHR CX	! 7	PUSH frmstr	! 27
.PUSH.CX	! 8	PUSH _PRINTF	! 28
CALL vecprint	! 9	SYS	! 29
MOV SP,BP	! 10	MOV -4(BP),frmatint	! 30
PUSH 0	! 11	1: MOV DI,(BX)(SI)	! 31
PUSH _EXIT	! 12	MOV -2(BP),DI	! 32
SYS	! 13	SYS	! 33
.SECT .DATA	! 14	INC SI	! 34
vec1: .WORD 3,4,7,11,3	! 15	LOOP 1b	! 35
frmstr: .ASCIZ "%s"	! 16	PUSH '\n'	! 36
frmatkop:	! 17	PUSH _PUTCHAR	! 37
.ASCIZ "L'array contiene "	! 18	SYS	! 38
frmatint: .ASCIZ "%d"	! 19	MOV SP,BP	! 39
		RET	! 40
			! 41

Figura C.16 Programma `arrayprt` prima del debugging.

Le righe dalla 27 alla 30 mostrano la stampa di una stringa, quelle dalla 31 alla 34 contengono la chiamata di sistema `printf` per un valore intero. Si noti che l'indirizzo della stringa viene impilato alla riga 27, mentre il valore intero viene copiato nello stack alla riga 33. In entrambi i casi l'indirizzo della stringa di formattazione è il primo argomen-

to della `PRINTF`. Le righe dalla 37 alla 39 mostrano come stampare un carattere usando la chiamata di sistema `putchar`.

Dopo aver assemblato il programma, ne lanciamo l'esecuzione con il comando

```
as88 arrayprt.s
```

e riceviamo un errore di operando alla riga 28 del file `arrayprt.s`. Questo file è prodotto dall'assemblatore combinando il file sorgente ai file inclusi ed è il vero input del processo di assemblaggio. Per individuare il contenuto della riga 28 dobbiamo aprire `arrayprt.s` in lettura, mentre non serve a nulla ispezionare `arrayprt.s`, perché i due file non corrispondono, in quanto il primo include il file d'intestazione alla prima riga. La riga 28 di `arrayprt.s` corrisponde alla riga 7 di `arrayprt.s` dal momento che il file d'intestazione, `../syscalnr.h`, contiene 21 righe.

Un modo semplice per visualizzare la riga 28 di `arrayprt.s` su di una macchina UNIX è lanciare il comando

```
head -28 arrayprt.s
```

che mostra le prime 28 righe del file `.s`. L'ultima riga è quella contenente l'errore. Così facendo (o in modo analogo con l'ausilio di un editor) notiamo che l'errore è alla riga 7 del sorgente originale, cioè in corrispondenza dell'istruzione `SHR`. Se confrontiamo la riga di codice con l'elemento corrispondente della Figura C.4 individuiamo il problema: abbiamo dimenticato d'indicare la lunghezza dello scorrimento. Una volta corretta la riga, diventa

```
SHR CX,1
```

È importante sottolineare che l'errore va corretto nel file sorgente originale `arrayprt.s` e *non* nel sorgente ausiliario `arrayprt.s`, rigenerato automaticamente a ogni chiamata dell'assemblatore.

Il nuovo tentativo di assemblare il codice sorgente dovrebbe andare a buon fine. Si può procedere con il lancio del tracer

```
t88 arrayprt
```

Durante l'esecuzione del tracer notiamo che il risultato del programma non corrisponde al vettore che si trova nel segmento dati. Il vettore contiene: 3, 4, 7, 11, 3 ma i valori visualizzati sono: 3, 1024, ... . Chiaramente c'è qualcosa che non va.

Per trovare l'errore facciamo ripartire il tracer, ma questa volta procediamo nella simulazione un'istruzione alla volta, esaminando lo stato della macchina appena prima della stampa del valore errato. Il valore da stampare è memorizzato alle righe 32 e 33; poiché la stampa visualizza un valore errato, è questo il punto dove rivolgere l'attenzione. Durante la seconda iterazione del ciclo, `SI` contiene un valore dispari, mentre dovrebbe contenere un numero pari, dato che indica parole, non byte. L'errore si trova alla riga 35: `SI` viene incrementato di 1, laddove andrebbe incrementato di 2. Per correggere il baco bisogna modificare la riga in questo modo

```
ADD SI,2
```

Dopo la correzione la lista dei numeri stampati risulta corretta.

Eppure c'è ancora un errore da scoprire. Al termine di `vecprint` il tracer visualizza un errore relativo al puntatore allo stack. Il controllo ovvio da effettuare in questo caso è verificare se il valore impilato sullo stack all'atto della chiamata di `vecprint` è uguale al valore che si trova in cima allo stack al momento della chiamata `RET` della riga 41. Il controllo evidenzia che i due valori sono diversi. La soluzione al problema consiste nel sostituire la riga 40 con le due righe:

```
ADD SP,10
POP BP
```

La prima istruzione rimuove dallo stack le 5 parole impilate durante l'esecuzione di `vecprint`, riportando in cima il valore di `BP` salvato alla riga 22. La pop di questo valore verso `BP` ripristina nel registro il valore che aveva prima della chiamata e fa emergere in cima allo stack l'indirizzo di ritorno corretto. Adesso il programma termina correttamente. Il debugging del codice in linguaggio assemblativo somiglia sicuramente più a un'arte che a una scienza, tuttavia risulta molto facilitato dall'uso del tracer rispetto alla "nuda" esecuzione in hardware.

### C.8.5 Manipolazione di stringhe e istruzioni su stringhe

Lo scopo principale di questo paragrafo è presentare le istruzioni iterative sulle stringhe. La Figura C.17 mostra due semplici programmi per la manipolazione di stringhe, `strncpy.s` e `reverspr.s`, contenuti nella directory `examples`. La Figura C.17(a) contiene una subroutine per la copia di una stringa che si avvale di un'altra subroutine, `stringpr.s`, che non mostriamo qui, ma che è reperibile nel file `stringpr.s`. Per assemblare un programma che comprende più subroutine contenute in file sorgenti diversi, bisogna elencarli tutti dopo il comando `as88`, a cominciare dal file sorgente contenente il programma principale. Quest'ultimo determina il nome del file eseguibile e del file `.s`. Per esempio, nel caso del programma della Figura C.17(a) bisogna digitare

```
as88 strncpy.s stringpr.s
```

Il programma della Figura C.17(b) stampa le stringhe in ordine inverso. Descriviamo ora i due programmi.

Per sottolineare il fatto che i numeri di riga sono solo commenti, nella Figura C.17(a) abbiamo numerato le righe a partire dalla prima etichetta, omettendo le righe che la precedevano. Il programma principale occupa le righe dalla 2 alla 8: per prima cosa richiama `strncpy` con due argomenti, la stringa sorgente `mesg2` e la stringa destinazione `mesg1`, perché copia la sorgente nella destinazione.

Diamo uno sguardo a `strncpy`, cominciando dalla riga 9. La subroutine si aspetta di trovare in cima allo stack gli indirizzi della destinazione e della sorgente. Nelle righe dalla 10 alla 13 si provvede a salvare i registri nello stack di modo che possano essere ripristinati successivamente con le righe dalla 27 alla 30. Alla riga 14 si effettua la copia di `SP` in `BP` come di consueto; adesso `BP` è pronto per essere usato come puntatore base per l'accesso agli argomenti. Ancora una volta il ripristino dello stack avviene mediante la copia di `BP` in `SP`, alla riga 26.

```

.SECT. TEXT
stcstart:
PUSH mesg1
PUSH mesg2
CALL stringcp
ADD SP,4
PUSH 0
PUSH 1
SYS
stringcp:
PUSH CX
PUSH SI
PUSH DI
PUSH BP
MOV BP,SP
MOV AX,0
MOV DI,10(BP)
MOV CX,-1
REPNZ SCASB
NEG CX
DEC CX
MOV SI,10(BP)
MOV DI,12(BP)
PUSH DI
REP MOVSB
CALL stringpr
MOV SP,BP
POP BP
POP DI
POP SI
POP CX
RET
.SECT .DATA
mesg1: .ASCIZ "Dacci un'occhiata\n"
mesg2: .ASCIZ "qrst\n"
.SECT .BSS
    ! 1 #include "../syscalnr.h" ! 1
    ! 2 start: MOV DI,str ! 2
    ! 3 PUSH AX ! 3
    ! 4 MOV BP,SP ! 4
    ! 5 PUSH _PUTCHAR ! 5
    ! 6 MOVB AL,'\n' ! 6
    ! 7 MOV CX,-1 ! 7
    ! 8 REPNZ SCASB ! 8
    ! 9 NEG CX ! 9
    ! 10 STD ! 10
    ! 11 DEC CX ! 11
    ! 12 SUB DI,2 ! 12
    ! 13 MOV SI,DI ! 13
    ! 14 1:LODSB ! 14
    ! 15 MOV (BP),AX ! 15
    ! 16 SYS ! 16
    ! 17 LOOP 1b ! 17
    ! 18 MOVB (BP),'\\n' ! 18
    ! 19 SYS ! 19
    ! 20 PUSH 0 ! 20
    ! 21 PUSH _EXIT ! 21
    ! 22 SYS ! 22
    ! 23 .SECT .DATA ! 23
    str: .ASCIZ "reverse\\n" ! 24
    (a) (b)

```

Figura C.17 (a) Copia di una stringa (stringcp.s). (b) Stampa di una stringa in ordine inverso (reverspr.s).

Il cuore della subroutine è l'istruzione REP MOVSB della riga 24. L'istruzione MOVSB trasferisce il byte puntato da SI nella locazione di memoria puntata da DI. SI e DI vengono poi incrementati di un'unità. L'istruzione REP crea un ciclo in cui, a ogni iterazione, si esegue MOVSB e si-decrementa CX di 1 dopo ogni trasferimento di un byte. Il ciclo termina quando CX raggiunge il valore 0.

Prima di poter eseguire REP MOVSB, bisogna però preparare i registri, come svolto nelle righe dalla 15 alla 22. L'indice sorgente SI viene copiato nella riga 21 dall'argomento che si trova sullo stack; l'indice destinazione DI è assegnato alla riga 22. Il calcolo del valore di CX è più difficile. Si tenga presente che il carattere di terminazione di una stringa è il byte zero. L'istruzione MOVSB non modifica il flag di zero, a differenza dell'istruzione SCASB (*SCAn Byte String*) che confronta il valore puntato da DI al contenuto di AL e incrementa DI al volo. Anche SCASB è un'istruzione iterabile come MOVSB.

Dunque alla riga 15 viene azzerato AX (e anche AL), alla riga 16 viene prelevato dallo stack il valore cui far puntare DI e alla riga 17 CX viene inizializzato a -1. La riga 18 contiene REP NZ SCASB, che effettua il confronto iterato e asserisce la flag zero quando rileva l'uguaglianza. A ogni iterazione del ciclo CX viene decrementato e il ciclo termina quando il flag di zero viene asserito, dal momento che REP NZ controlla sia il valore di CX, sia il contenuto di quel flag. Il numero d'iterazioni di MOVSB si ottiene sottraendo il valore corrente di CX al suo valore iniziale -1, come effettuato alle righe 19 e 20.

Purtroppo si rendono necessarie ben due istruzioni iterative, ma è il prezzo che bisogna pagare per la scelta progettuale secondo cui le istruzioni di trasferimento non possono modificare i codici di condizione. I registri indice devono essere incrementati durante i cicli, e a tal fine è necessario azzerare il flag della modalità d'incremento.

Le righe 23 e 25 stampano la stringa copiata attraverso la chiamata della subroutine *stringpr*, anch'essa contenuta nella directory *examples*. Il suo comportamento è di facile comprensione, perciò ne omettiamo la trattazione.

Il programma della Figura C.17(b) per la stampa invertita comincia con l'usuale riga d'inclusione delle costanti per le chiamate di sistema. La riga 3 impila sullo stack un dato fittizio, mentre la riga 4 fa sì che BP punti alla cima dello stack. Il programma stampa i caratteri ASCII uno per volta, perciò il valore numerico di \_PUTCHAR viene posto in cima allo stack. BP punta al carattere da stampare al momento della chiamata di SYS.

Le righe 2, 6 e 7 preparano i registri DI, AL e CX per l'istruzione SCASB. Il registro per la lunghezza e per l'indice di destinazione sono caricati in modo analogo a quanto visto nella routine per la copia di una stringa, ma il valore di AL è ora il carattere di fine riga e non il valore 0. Così facendo, l'istruzione SCASB confronterà i caratteri della stringa *str* con \n e non con 0, e assegnerà il flag zero non appena troverà una corrispondenza.

REP SCASB incrementa il registro DI perciò, dopo aver trovato la corrispondenza, l'indice destinazione punta al carattere zero che segue il carattere di fine riga. La riga 12 decremente DI di 2 di modo che punti all'ultima lettera dell'ultima parola della stringa. Per produrre il risultato desiderato si può procedere percorrendo la stringa in ordine inverso e stampando un carattere alla volta, perciò il flag della modalità d'incremento viene asserito alla riga 10 in modo da invertire la modalità d'incremento dei registri indice adottata dalle istruzioni sulle stringhe. L'istruzione LODSB alla riga 14 copia il carattere in AL e la riga 15 lo inserisce nella locazione dello stack adiacente a \_PUTCHAR, così che venga stampato dall'esecuzione di SYS.

Le istruzioni delle righe 18 e 19 stampano un carattere di fine riga e il programma termina con la chiamata \_EXIT nel modo usuale.

Questa versione del programma contiene però un baco che può essere individuato eseguendolo con il tracer un'istruzione alla volta.

Il comando /str inserisce la stringa *str* nell'area dati del tracer. Il valore numerico dell'indirizzo dei dati è visibile, perciò è possibile monitorare la progressione delle posizioni dei registri indice all'interno dei dati rispetto all'indirizzo della stringa.

Il baco si manifesterà solo dopo aver premuto più volte il tasto Invio. Possiamo usare i comandi del tracer per raggiungere il problema più velocemente. Dopo aver avviato il tracer gli si invia il comando 13 per posizionarsi nel mezzo del ciclo. Il comando b inserisce un breakpoint in corrispondenza della riga 15. Dopo due invii a capo vediamo

apparire la lettera e (ultimo carattere della stringa “reverse”) nell’area di output. Il comando *r* ordina al tracer di continuare l’esecuzione fino al prossimo breakpoint o fino alla fine del processo. In questo modo possiamo scorrere le lettere imparando ogni volta il comando *r* finché non ci avviciniamo al problema. Solo allora riprendiamo l’esecuzione un’istruzione alla volta, fino a raggiungere l’istruzione critica.

In alternativa è possibile inserire il breakpoint a una riga specifica, ma a tal fine bisogna tenere presente che è stato incluso il file *syscalnr.h* e che i numeri di riga vanno spostati di 21. Di conseguenza il breakpoint alla riga 15 può essere inserito con il comando *36b*; questa non è una scelta molto elegante, è preferibile usare l’etichetta globale *start* della riga 2 e impartire il comando */start +13b* che inserisce il breakpoint nello stesso punto e permette di trascurare la dimensione del file d’intestazione.

### C.8.6 Tabella di salto

Molti linguaggi di programmazione mettono a disposizione le istruzioni *case* o *switch* per selezionare una destinazione di salto tra numerose alternative in base al valore di una certa variabile. A volte queste diramazioni “a più vie” (*multiway branch*) sono necessarie anche nei programmi in linguaggio assemblativo. Si pensi per esempio a un insieme di chiamate di sistema invocabili tramite un’unica trap *SYS*. Il programma *jumptbl.s* della Figura C.18 mostra un modo per utilizzare questo costrutto nell’assemblatore dell’8088.

Il programma comincia con la stampa della stringa che si trova all’etichetta *strt* e che invita l’utente a immettere una cifra ottale (righe dalla 4 alla 7), quindi legge un carattere dallo standard input (righe 8 e 9). Se il valore di *AX* è minore di 5 il programma lo interpreta come un marcatore di terminazione di file e salta all’etichetta 8 della riga 22, provocando l’uscita del programma con codice d’uscita 0.

In caso contrario, il carattere inserito in *AL* viene esaminato nel modo seguente: se è minore del carattere 0 viene interpretato come uno spazio bianco e viene ignorato saltando alla riga 13, per ricevere un altro carattere. Gli input maggiori della cifra 9 sono considerati errati e vengono mappati nel carattere ASCII dei due punti (:) che segue il carattere 9 nella sequenza dei codici della tabella ASCII.

La riga 17 copia in *BX* il contenuto di *AX*, che a questo punto contiene un valore compreso tra 0 e i due punti. L’istruzione *AND* della riga 18 maschera tutti i bit di *BX* tranne i quattro meno significativi, producendo un risultato che va da 0 a 10 (poiché la cifra 0 corrisponde al codice ASCII 0x30). Il valore di *BX* viene moltiplicato per 2 alla riga 19 con uno scorrimento a sinistra, dato che ci apprestiamo a indirizzare una tabella di parole e non di byte.

La riga 20 contiene un’istruzione di chiamata. L’indirizzo effettivo, ottenuto sommando *BX* al valore numerico dell’etichetta *tbl*, viene caricato nel program counter.

Il programma sceglie una delle dieci subroutine a seconda del carattere ricevuto in ingresso dallo standard input. Ciascuna di queste subroutine impila sullo stack l’indirizzo di un messaggio e salta alla chiamata della subroutine di sistema *\_PRINTF* condivisa da tutte e dieci.

Per capire come funziona il programma bisogna sapere che le istruzioni *JMP* e *CALL* carcano un certo indirizzo del segmento di testo nel PC. Questo indirizzo non è altro che

un numero, poiché il processo di assemblaggio sostituisce tutti gli indirizzi con i loro valori binari. Alla riga 50 questi valori binari vengono utilizzati per inizializzare un array nel segmento dati. L’array che comincia alla locazione *tbl* contiene gli indirizzi *rout0*, *rout1*, *rout2*, e così via, ciascuno lungo due byte. La necessità di usare indirizzi di due byte giustifica lo scorrimento di un bit che abbiamo incontrato alla riga 19. Un array di questo tipo si dice *tabella di salto* (*dispatch table*).

```
#include "../syscalnr.h"
.SECT .TEXT
jumpstr:    ! 1
    PUSH str
    MOV BP,SP
    PUSH _PRINTF
    SYS
    PUSH _GETCHAR
1:   SYS
    CMP AX,5
    JL 8f
    CMPB AL,'0'
    JL 1b
    CMPB AL,'9'
    JLE 2f
    MOVB AL,'9'+1
2:   MOV BX,AX
    AND BX,0xF
    SAL BX,1
    CALL tbl(BX)
    JMP 1b
8:   PUSH 0
    PUSH _EXIT
    SYS
    ! 24

.rout0: MOV AX,mes0
        JMP 9f
.rout1: MOV AX,mes1
        JMP 9f
.rout2: MOV AX,mes2
        JMP 9f
.rout3: MOV AX,mes3
        JMP 9f
.rout4: MOV AX,mes4
        JMP 9f
.rout5: MOV AX,mes5
        JMP 9f
.rout6: MOV AX,mes6
        JMP 9f
.rout7: MOV AX,mes7
        JMP 9f
.rout8: MOV AX,mes8
        JMP 9f
.erout: MOV AX,emes
.9:   PUSH AX
      PUSH _PRINTF
      SYS
      ADD SP,4
      RET
      ! 48

.SECT .DATA
tbl: WORD rout0,rout1,rout2,rout3,rout4,rout5,rout6,rout7,rout8,rout8,erout
mes0: .ASCIZ "La cifra è uno zero.\n"
mes1: .ASCIZ "La cifra è un uno.\n"
mes2: .ASCIZ "La cifra è un due.\n"
mes3: .ASCIZ "La cifra è un tre.\n"
mes4: .ASCIZ "La cifra è un quattro.\n"
mes5: .ASCIZ "La cifra è un cinque.\n"
mes6: .ASCIZ "La cifra è un sei.\n"
mes7: .ASCIZ "La cifra è un sette.\n"
mes8: .ASCIZ "La cifra non è una cifra ottale valida.\n"
emes: .ASCIZ "Questa non è una cifra ottale.\n"
strt: .ASCIZ "Inserisci una cifra ottale valida seguita da invio. Per uscire inserisci un carattere di fine file.\n"
```

Figura C.18 Programma che implementa una diramazione a più vie usando una tabella di salto.

Osserviamo il funzionamento di queste routine descrivendo la routine *erout* che occupa le righe dalla 43 alla 48 e che gestisce il caso di un carattere esterno all’intervallo delle cifre. Alla riga 43 viene posto in cima allo stack l’indirizzo del messaggio (che si trova

2 e imparire il comando `/start + 13b` che inserisce il breakpoint nello stesso punto e permette di trascurare la dimensione del file d'intestazione.

### C.8.6 Tabella di salto

Molti linguaggi di programmazione mettono a disposizione le istruzioni `case` o `switch` per selezionare una destinazione di salto tra numerose alternative in base al valore di una certa variabile. A volte queste diramazioni "a più vie" (*multiway branch*) sono necessari anche nei programmi in linguaggio assemblativo. Si pensi per esempio a un insieme di chiamate di sistema invocabili tramite un'unica trap `SYS`. Il programma `jumptbl.s` della Figura C.18 mostra un modo per utilizzare questo costrutto nell'assemblatore dell'8088.

Il programma comincia con la stampa della stringa che si trova all'etichetta `strt` e che invita l'utente a immettere una cifra ottale (righe dalla 4 alla 7), quindi legge un carattere dallo standard input (righe 8 e 9). Se il valore di `AX` è minore di 5 il programma lo interpreta come un marcitore di terminazione di file e salta all'etichetta 8 della riga 22, provando l'uscita del programma con codice d'uscita 0.

In caso contrario, il carattere inserito in `AL` viene esaminato nel modo seguente: se è minore del carattere 0 viene interpretato come uno spazio bianco e viene ignorato saltando alla riga 13, per ricevere un altro carattere. Gli input maggiori della cifra 9 sono considerati errati e vengono mappati nel carattere ASCII dei due punti (:) che segue il carattere 9 nella sequenza dei codici della tabella ASCII.

La riga 17 copia in `BX` il contenuto di `AX`, che a questo punto contiene un valore compreso tra 0 e i due punti. L'istruzione `AND` della riga 18 maschera tutti i bit di `BX` tranne i quattro meno significativi, producendo un risultato che va da 0 a 10 (poiché la cifra 0 corrisponde al codice ASCII 0x30). Il valore di `BX` viene moltiplicato per 2 alla riga 19 con uno scorrimento a sinistra, dato che ci apprestiamo a indirizzare una tabella di parole e non di byte.

La riga 20 contiene un'istruzione di chiamata. L'indirizzo effettivo, ottenuto sommando `BX` al valore numerico dell'etichetta `tbl`, viene caricato nel program counter.

Il programma sceglie una delle dieci subroutine a seconda del carattere ricevuto in ingresso dallo standard input. Ciascuna di queste subroutine impila sullo stack l'indirizzo di un messaggio e salta alla chiamata della subroutine di sistema `_PRINTF` condivisa da tutte e dieci.

Per capire come funziona il programma bisogna sapere che le istruzioni `JMP` e `CALL` carcano un certo indirizzo del segmento di testo nel PC. Questo indirizzo non è altro che un numero, poiché il processo di assemblaggio sostituisce tutti gli indirizzi con i loro valori binari. Alla riga 50 questi valori binari vengono utilizzati per inizializzare un array nel segmento dati. L'array che comincia alla locazione `tbl` contiene gli indirizzi `rout0`, `rout1`, `rout2`, ecc., ciascuno lungo due byte. La necessità di usare indirizzi di due byte giustifica lo scorrimento di un bit che abbiamo incontrato alla riga 19. Un array di questo tipo si dice **tabella di salto** (*dispatch table*).

Osserviamo il funzionamento di queste routine descrivendo la routine `erout` che occupa le righe dalla 43 alla 48 e che gestisce il caso di un carattere esterno all'intervallo delle cifre. Alla riga 43 viene posto in cima allo stack l'indirizzo del messaggio (che si trova in `AX`), poi viene impilato il numero della chiamata di sistema `_PRINTF`, si effettua la chiamata, si ripristina lo stack e infine la routine restituisce il controllo al chiamante. Anche le altre nove routine da `rout0` a `rout8` carcano in `AX` l'indirizzo del loro messaggio privato, poi saltano alla seconda riga di `erout` per stampare il messaggio e terminare l'esecuzione della subroutine.

```
#include "../syscalnr.h"
.SECT .TEXT
jumpstrt:
    PUSH strt
    MOV BP,SP
    PUSH _PRINTF
    SYS
    PUSH _GETCHAR
1:   SYS
    CMP AX,5
    JL 8f
    CMPB AL,'0'
    JL 1b
    CMPB AL,'9'
    JLE 2f
    MOVB AL,'9'+1
2:   MOV BX,AX
    AND BX,0xf
    SAL BX,1
    CALL tbl(BX)
    IMP 1b
8:   PUSH 0
    PUSH _EXIT
    SYS
    ! 1
    ! 2
    ! 3
    ! 4
    ! 5
    ! 6
    ! 7
    ! 8
    ! 9
    ! 10
    ! 11
    ! 12
    ! 13
    ! 14
    ! 15
    ! 16
    ! 17
    ! 18
    ! 19
    ! 20
    ! 21
    ! 22
    ! 23
    ! 24
rout0: MOV AX,mes0
        JMP 9f
rout1: MOV AX,mes1
        JMP 9f
rout2: MOV AX,mes2
        JMP 9f
rout3: MOV AX,mes3
        JMP 9f
rout4: MOV AX,mes4
        JMP 9f
rout5: MOV AX,mes5
        JMP 9f
rout6: MOV AX,mes6
        JMP 9f
rout7: MOV AX,mes7
        JMP 9f
rout8: MOV AX,mes8
        JMP 9f
erout: MOV AX,emes
9:   PUSH AX
    PUSH _PRINTF
    SYS
    ADD SP,4
    RET
    ! 25
    ! 26
    ! 27
    ! 28
    ! 29
    ! 30
    ! 31
    ! 32
    ! 33
    ! 34
    ! 35
    ! 36
    ! 37
    ! 38
    ! 39
    ! 40
    ! 41
    ! 42
    ! 43
    ! 44
    ! 45
    ! 46
    ! 47
    ! 48
    ! 49
    ! 50
    ! 51
    ! 52
    ! 53
    ! 54
    ! 55
    ! 56
    ! 57
    ! 58
    ! 59
    ! 60
    ! 61
.SECT .DATA
tbl: .WORD rout0,rout1,rout2,rout3,rout4,rout5,rout6,rout7,rout8,rout8,erout
mes0: .ASCIZ "La cifra è uno zero.\n"
mes1: .ASCIZ "La cifra è un uno.\n"
mes2: .ASCIZ "La cifra è un due.\n"
mes3: .ASCIZ "La cifra è un tre.\n"
mes4: .ASCIZ "La cifra è un quattro.\n"
mes5: .ASCIZ "La cifra è un cinque.\n"
mes6: .ASCIZ "La cifra è un sei.\n"
mes7: .ASCIZ "La cifra è un sette.\n"
mes8: .ASCIZ "La cifra non è una cifra ottale valida.\n"
emes: .ASCIZ "Questa non è una cifra ottale.\n"
strt: .ASCIZ "Inserisci una cifra ottale valida seguita da invio. Per uscire inserisci un carattere di fine file.\n"
```

Figura C.18 Programma che implementa una diramazione a più vie usando una tabella di salto.

Per abituarsi all'uso delle tabelle di salto, conviene simulare il programma con il tracer e osservarne il comportamento per diversi caratteri di input. Come esercizio, esortiamo il lettore a modificare il programma perché risponda a ogni input con un'azione adeguata: per esempio potrebbe stampare un messaggio d'errore tutte le volte che viene inserito un carattere che non corrisponde a una cifra ottale.

### C.8.7 File: accesso diretto e accesso bufferizzato

La Figura C.19 mostra il programma `InFilBuf.s` che illustra l'I/O di file con accesso diretto. Un file è pensato come un insieme di righe che possono avere lunghezze diverse. Il programma comincia con la lettura del file e costruisce una tabella il cui elemento `n` contiene la locazione del file in cui inizia la riga `n`. Se poi giunge una richiesta per una riga del file, si ac-

```

#include "./syscalnr.h" ! 1    PUSH_EXIT      ! 43    PUSH buf          ! 85
bufsiz = 512           ! 2    PUSH_EXIT      ! 44    PUSH(fildes)     ! 86
.SECT .TEXT            ! 3    SYS             ! 45    PUSH_READ        ! 87
inbufst:               ! 4    3: CALL getnum   ! 46    SYS             ! 88
    MOV BP,SP          ! 5    CMP AX,0       ! 47    ADD SP,8         ! 89
    MOV DI,linein      ! 6    JLE 8f          ! 48    MOV CX,AX       ! 90
    PUSH _GETCHAR      ! 7    MOV BX,(curlin) ! 49    ADD BX,CX       ! 91
1: SYS                ! 8    CMP BX,0       ! 50    MOV DI,buf        ! 92
    CMPB AL,'\n'       ! 9    JLE 7f          ! 51    RET             ! 93
    JE 1f              ! 10   CMP BX,(count) ! 52
    JG 7f              ! 11   getnum:        ! 53
        MOV DI,linein  ! 54
        PUSH _GETCHAR  ! 55
1: SYS                ! 56
    CMPB AL,'\n'       ! 57
    JL 9b              ! 58
    JE 1f              ! 59
    STOSB             ! 60
    PUSH AX            ! 61
    PUSH _LSEEK          ! 62
    1: MOVB (DI),''    ! 63
    PUSH curlin        ! 64
    PUSH numfmt        ! 65
    PUSH linea          ! 66
    PUSH _SSCANF        ! 67
    SYS                ! 68
    ADD SP,10          ! 69
    RET                ! 70
    .SECT .DATA         ! 71
    errmess:           ! 72
    .ASCIZ "Apertura di %s non riuscita\n" ! 73
    numfmt:            ! 74
    .ASCIZ "%d"         ! 75
    scanner:           ! 76
    .ASCIZ "Digitare un numero.\n" ! 77
    .ALIGN 2            ! 78
    .SECT .BSS          ! 79
    linea: .SPACE 80    ! 80
    fildes: .SPACE 2    ! 81
    linh: .SPACE 8192   ! 82
    curlin: .SPACE 4    ! 83
    buf: .SPACE bufsiz+2 ! 84
    count: .SPACE 2     ! 85
9: MOV SP,BP          ! 86
    PUSH linea          ! 87
    PUSH errmess        ! 88
    PUSH _PRINTF        ! 89
    SYS                ! 90
    7: PUSH 0            ! 91
    PUSH_EXIT          ! 92
    fillbuf:           ! 93
    PUSH bufsiz         ! 94

```

Figura C.19 Programma per l'accesso diretto e per l'accesso bufferizzato di file.

cede all'elemento corrispondente della tabella e si legge la riga con le chiamate di sistema `lseek` e `read`. Il programma accetta come primo dato il nome del file immesso da standard input. Il codice è diviso in porzioni abbastanza indipendenti che possono essere modificate per assolvere ad altre funzioni.

Le prime cinque righe di codice servono a definire i numeri delle chiamate di sistema e la dimensione del buffer, e ad assegnare la cima dello stack al puntatore base, come di consueto. Le righe dalla 6 alla 13 leggono il nome del file dallo standard input e lo memorizzano come stringa all'etichetta `linein`. Se il nome del file non è seguito da un carattere di fine riga, viene visualizzato un messaggio d'errore e il processo termina con un codice di stato diverso da zero (righe da 38 a 45). Si noti che l'indirizzo del nome del file viene impilato alla riga 39, mentre l'indirizzo del messaggio d'errore alla riga 40. Osserviamo che il messaggio d'errore (riga 113) contiene la combinazione di caratteri `%s`, cioè una richiesta per

una stringa secondo il formato di `_PRINTF`. La richiesta viene soddisfatta dall'inserimento della stringa `linein` al posto di `%s`.

Se il nome del file può essere copiato senza problemi, allora si procede con l'apertura del file dalla riga 14 alla 20. Se la chiamata `open` non ha successo, restituisce un valore negativo che provoca il salto all'etichetta 9 della riga 28 per la stampa di un messaggio d'errore. Altrimenti la `open` restituisce il descrittore di file che viene memorizzato nella variabile `fildes`. Il descrittore di file serve per le successive chiamate di `read` e `lseek`.

Poi si procede con la lettura del file a blocchi di 512 byte, memorizzati nel buffer `buf`. Il buffer è stato allocato in un'area di memoria che eccede di due byte la dimensione di blocco per puro scopo dimostrativo: la dimensione può essere specificata tramite un'espressione che combina una costante simbolica a un intero (riga 123). Analogamente, alla riga 21 viene assegnato a `SI` il secondo elemento dell'array `linh`, perciò resta una parola di valore 0 tra l'inizio dell'array e la locazione puntata da `SI`. Il registro `BX` serve a contenere la locazione del primo carattere del file non ancora letto, perciò viene inizializzato a 0 prima che il buffer venga riempito alla riga 22.

Il riempimento del buffer è gestito dalla routine `fillbuf` che occupa le righe dalla 83 alla 93. La chiamata di sistema, effettuata dopo l'inserzione degli argomenti sullo stack, salva in `AX` il numero di byte effettivamente letti. Questo valore viene copiato in `CX`, che serve a contenere il numero di caratteri non letti ancora presenti nel buffer. `BX` contiene invece la posizione nel file del primo carattere non letto, perciò alla riga 91 si somma `CX` a `BX`. La riga 92 assegna a `DI` la prima posizione del buffer per preparare la sua scansione in cerca del successivo carattere di fine riga.

In seguito al ritorno da `fillbuf`, la riga 24 verifica che sia stato letto davvero qualcosa. In caso contrario si esce dalla parte di lettura bufferizzata e si salta alla riga 25, dove comincia la seconda parte del programma.

Siamo pronti per la scansione del buffer. La riga 26 carica il simbolo `\n` in `AL`, la riga 27 ricerca questo valore nel buffer leggendolo iterativamente con il ciclo `REP SCASB`. Ci sono due condizioni di uscita dal ciclo: `CX` raggiunge il valore 0 oppure uno dei valori è un carattere di fine riga. Se il flag zero è asserito vuol dire che l'ultimo simbolo scandito è `\n` e che la posizione del simbolo corrente (quello successivo al carattere di fine riga) deve essere memorizzato nell'array `linh`. Quindi viene incrementato il contatore `count`, viene calcolata la posizione del file in base al contenuto di `BX` e `CX` stabilisce il numero di caratteri ancora disponibili (righe dalla 29 alla 31). Le righe dalla 32 alla 24 effettuano la memorizzazione vera e propria: il contenuto dei registri `SI` e `DI` viene scambiato prima e dopo l'istruzione `STOS` perché questa considera `DI` come destinazione e `SI` come sorgente. Le righe 35, 36 e 37 verificano se il buffer contiene altri dati da trattare e saltano in base al valore di `CX`.

Una volta raggiunto il carattere di terminazione del file, abbiamo costruito una lista completa contenente le posizioni del primo carattere di ogni riga. Dato che `linh` comincia con la parola 0, sappiamo che la prima riga comincia all'indirizzo 0, mentre la seconda riga comincia alla posizione contenuta nella parola che si trova in `linh + 2`, e così via. La lunghezza della riga `n` è data dall'indirizzo d'inizio della riga `n + 1` meno la posizione iniziale della riga `n`.

La parte restante del programma serve per leggere il numero di una riga, copiare la riga nel buffer e visualizzarla tramite la chiamata `write`. L'array `linh` contiene tutte le informazioni necessarie, dato che il suo `n`-esimo elemento contiene la locazione iniziale della riga `n`. Se il numero di riga richiesto è 0 o se è maggiore del numero di righe presenti, il programma termina saltando all'etichetta 7.

Questa porzione di programma comincia con la chiamata alla subroutine *getnum* alla riga 46. La routine legge una riga dallo standard input e la memorizza nel buffer *linein* (righe dalla 95 alla 103). Segue la preparazione della chiamata *SSCANF*, i cui argomenti vengono impilati in ordine inverso: si effettua il push di *curlin*, che può contenere un valore intero, quindi c'è il push dell'indirizzo della stringa di formattazione *numfmt* e infine viene impilato l'indirizzo del buffer *linein* contenente il numero in notazione decimale. La subroutine di sistema *SSCANF* inserisce in *curlin* il valore binario se l'operazione è possibile, altrimenti restituisce in *AX* il valore 0. La riga 48 effettua il test del valore restituito: in caso d'insuccesso il programma genera un messaggio d'errore saltando all'etichetta 8.

Se la subroutine *getnum* restituisce in *curlin* un valore intero corretto, questo viene copiato in *BX* e poi valutato nelle righe dalla 49 alla 53 per verificare se corrisponde a un numero di riga valido; in caso contrario il programma termina con la chiamata di *EXIT*.

A questo punto bisogna individuare la terminazione della riga selezionata e il numero di byte da leggere: a tal fine moltiplichiamo *BX* per 2 per mezzo dello scorrimento verso sinistra *SHL*. La posizione nel file della riga selezionata viene copiata in *AX* alla riga 55, mentre la posizione della riga successiva viene conservata in *CX* perché verrà usata in seguito per calcolare il numero di byte della riga corrente.

L'accesso diretto a un file si realizza tramite la chiamata *lseek* cui va specificato l'offset della posizione cui si vuole accedere direttamente. L'offset è specificato in base all'inizio del file, perciò alla riga 57 viene impilato un argomento con valore 0. L'argomento successivo è l'offset nel file di tipo long (cioè un intero di 32 bit), dunque impiliamo prima una parola con valore 0 e poi il valore di *AX* (righe 58 e 59) per formare un intero di 32 bit. In seguito impiliamo il descrittore di file e il codice di *LSEEK* prima di effettuare la chiamata della riga 62. Il risultato di *LSEEK* è la posizione corrente del file, memorizzata nella coppia di registri *DX:AX*. Se questo numero ci sta tutto in una parola (il che è sicuramente vero per file più piccoli di 65.536 byte), l'indirizzo è contenuto in *AX*, perciò sottraendolo da *CX* otteniamo il numero di byte da leggere per copiare la riga nel buffer.

Il resto del programma è di facile comprensione. Le righe dalla 64 alla 68 leggono la riga del file, le righe dalla 70 alla 72 scrivono la riga sullo standard output (che ha descrittore di file 1). Si noti che, dopo il ripristino parziale della riga 69, lo stack contiene ancora il contatore *count* e l'indirizzo del buffer. La riga 73 ripristina definitivamente il puntatore allo stack e, saltando all'etichetta 3, si è pronti per cominciare un'altra iterazione di *getnum*.

## Ringraziamenti

L'assemblatore utilizzato in questa appendice fa parte della raccolta di strumenti "Amsterdam Compiler Kit", disponibile all'indirizzo web [www.cs.vu.nl/ack](http://www.cs.vu.nl/ack). Ringraziamo coloro che hanno partecipato alla sua progettazione iniziale: Johan Stevenson, Hans Schaminee e Hans de Vries. Siamo debitori a Ceriel Jacobs che ha gestito lo sviluppo di questo pacchetto software e ci ha aiutati ad adattarlo più e più volte perché soddisfacesse i requisiti del nostro corso universitario. Siamo riconoscenti a Elth Ogston per la lettura del manoscritto e la verifica d'esempi ed esercizi.

Vogliamo ringraziare anche Robbert van Renesse, progettista del tracer di PDP-11, e Jan-Mark Wams, progettista del tracer di Motorola 68000. Molte delle loro idee sono entrate a far parte del progetto del tracer descritto in questa appendice. Inoltre, vogliamo ringraziare i tanti insegnanti e amministratori di sistema che ci hanno assistito durante la frequentazione dei molti corsi di programmazione in linguaggio assemblativo che abbiamo seguito negli anni passati.

## Problemi

- Dopo l'esecuzione dell'istruzione *MOV AX, 702*, qual è il valore (decimale) contenuto nei registri *AH* e *AL*?
- Il registro *CS* contiene il valore 4. Qual è l'intervallo d'indirizzi di memoria assoluti occupato dal segmento di codice?
- Qual è il più grande indirizzo di memoria accessibile dall'8088?
- Sia *CS* = 40, *DS* = 8000 e *IP* = 20.
  - Qual è l'indirizzo assoluto dell'istruzione successiva?
  - Se viene eseguita *MOV AX, (2)*, quale parola di memoria viene caricata in *AX*?
- Si immagini una subroutine con tre argomenti che viene chiamata con l'usuale sequenza: il chiamante impila i tre argomenti sullo stack in ordine inverso e quindi esegue l'istruzione *CALL*. Il chiamato salva il vecchio *BP* e inizializza il nuovo *BP* in modo che punti al suo vecchio valore. Quindi decrementa il puntatore allo stack in modo da allocare spazio sufficiente per le variabili locali. Date queste convenzioni, si indichi l'istruzione necessaria per copiare il primo argomento della subroutine nel registro *AX*.
- Nella Figura C.1 viene usata l'espressione *de - hw*, cioè la differenza di due etichette, come operando di un'istruzione. È possibile immaginare una circostanza in cui sarebbe sensato usare come operando la somma di due etichette, per esempio *de + hw*? Si motivi la risposta.
- Si scriva il codice assemblativo per il calcolo dell'espressione:  
*x = a + b + 2*  
x = a + b + 2
- Una funzione C viene invocata con l'istruzione:  
*foobar(x, y);*  
Si scriva il codice assemblativo che realizza la stessa chiamata.
- Si scriva un programma in linguaggio assemblativo che accetti in ingresso espressioni contenenti un intero, un operando e un altro intero, e che stampi in uscita il valore dell'espressione. Il programma deve gestire gli operatori *+, -, × e /*.

6. Nella Figura C.1 viene usata l'espressione  $de - hw$ , cioè la differenza di due etichette, come operando di un'istruzione. È possibile immaginare una circostanza in cui sarebbe sensato usare come operando la somma di due etichette, per esempio  $de + hw$ ? Si motivi la risposta.
7. Si scriva il codice assemblativo per il calcolo dell'espressione:  
 $x = a + b + 2$
8. Una funzione C viene invocata con l'istruzione:  
`foobar(x, y);`  
Si scriva il codice assemblativo che realizza la stessa chiamata.
9. Si scriva un programma in linguaggio assemblativo che accetti in ingresso espressioni contenenti un intero, un operando e un altro intero, e che stampi in uscita il valore dell'espressione. Il programma deve gestire gli operatori  $+, -, \times$  e  $/$ .

## Indice analitico

### A

- Access Control List (ACL), 510  
Accumulatore, 19, 708  
ACK (*vedi* Amsterdam Compiler Kit)  
ACL (*vedi* Access Control List)  
Acorn Archimedes, 47  
ADSL (*vedi* Asymmetric DSL)  
Advanced Microcontroller Bus Architecture (AMBA), 582  
Advanced Programmable Interrupt Controller (APIC), 212  
AGP, bus, 224  
Aiken, Howard, 16  
Albero, 629  
Algebra booleana minimale, 154  
Algebra di Boole, 151-162  
Algoritmi di paginazione, 455-457  
FIFO, 457  
LRU, 456  
Algoritmo, 8  
Algoritmo di allattamento a richiesta, 454  
Algoritmo first fit, 462  
Algoritmo First-In First-Out (FIFO), 456  
Algoritmo Least Recently Used (LRU), 319, 455  
Allocazione in scrittura, 320  
Alpha, 14, 27  
ALU (*vedi* Arithmetic Logic Unit)

- AMBA (*vedi* Advanced Microcontroller Bus Architecture)  
American Standard Code for Information Interchange (ASCII), 141  
Ampiezza del bus, 196-197  
Amsterdam Compiler Kit, 727  
Analytical engine, 15, 27  
Anello, 577, 629  
APIC (*vedi* Advanced Programmable Interrupt Controller)  
Apparecchiatura per le telecomunicazioni, 130-138  
Apple II, 25  
Apple Lisa, 25  
Apple Macintosh, 26  
Apple Newton, 14, 28, 47  
Application Programming Interface (API), 495  
Application-Specific Integrated Circuit (ASIC), 586  
Arbitraggio del bus, 202-205  
Arbitro del bus, 113  
Architettura a tre bus, 296  
Architettura degli elaboratori, 8, 77  
Architettura di un computer, 8  
Architettura Harvard, 85  
Architettura IA-32, 432  
difetti, 432

Architettura load/store, 365  
 Architettura superscalare, 67-69  
 architettura x86, 41  
 Arduino, 35  
 Area dei metodi, 267  
 Argomenti, 722  
 Arithmetic Logic Unit (ALU), 5, 56-57, 172  
 Aritmetica binaria, 690  
 Aritmetica satura, 567  
 ARM  
     bi-endian, 364  
     formati d'istruzione, 378-380  
     indirizzamento, 394  
     introduzione, 46-49  
     istruzioni, 410-412  
     livello ISA, 363-365  
     memoria virtuale, 468-471  
     microarchitettura dell'OMAP 4430, 341-343  
     tipi di dati, 370  
 ARM, 46 (*vedi anche* OMAP 4430)  
 ARM v7, 356, 365  
 Arrotondamento, 695  
 As88, 727-732  
 ASCII (*vedi* American Standard Code for Information Interchange)  
 ASIC (*vedi* Application Specific Integrated Circuit)  
 Assemblaggio di memoria, 86  
 Assemblatore a due passate, 537-545  
 Assemblatore, 7, 526, 530, 537, 726  
     8088, 731  
 Assemblatore, prima passata, 538-542  
 Assemblatore, seconda passata, 542-544  
 Asymmetric DSL, 133  
 AT attachment disk, 93  
 ATA packet interface, 93  
 Atanasoff, John, 16  
 ATmega168, 219-221, 346, 348  
     formati d'istruzione, 380  
     istruzioni, 412-413  
     livello ISA, 366-368  
     microarchitettura, 346-348  
     modalità d'indirizzamento, 394-395  
     tipi di dati, 371  
 Atmel ATmega168 (*vedi* ATmega168)  
 Attesa attiva, 405

Autoincremento, 715  
 Automa a stati finiti, 299, 325  
 Auto-modificante, programma, 384  
 AVR, 49-50  
**B**  
 Babbage, Charles, 15, 27  
 Banda larga, 133  
 Bardeen, John, 19  
 Base, 152, 683  
 Basic Input Output System (BIOS), 92  
 Baud, 132  
 Best fit, algoritmo, 462  
 Big Endian, memoria, 77-78  
 Binario, sistema di numerazione, 683  
 BIOS (*vedi* Basic Input Output System)  
 Bit, 74, 684  
 Bit di parità, 79  
 Bit map, 369  
 Bit presente/assente, 451  
 Blocchi di memoria, 450  
 Blocco delle variabili locali, 265  
 Blocco elementare, 332  
 Blocco indiretto, 506  
 Blocco indiretto doppio, 506  
 Blocco indiretto triplo, 498  
 BlueGene, 631-635  
 BlueGene/L, 631  
 BlueGene/P, 631  
 BlueGene/P e Red Storm, confronto, 631, 639  
 Blu-ray, 111  
 Boole, George, 154  
 Branch Target Buffer (BTB), 338  
 Brattain, Walter, 19  
 Breakpoint, 734  
 BSS (block started by symbol), 727  
 BTB (*vedi* Branch Target Buffer)  
 Buffer di prefetch, 65  
 Buffer di scrittura, OMAP 4430, 343  
 Buffer invertente, 183  
 Buffer non invertente, 183  
 Bundle, Itanium 2, 435  
 Burroughs B5000, 14, 22  
 Bus, 20, 55, 111-114, 191-208, 221-239  
     asincrono, 201-202  
     Core i7, 213-215

di sistema, 194  
 EISA, 113  
 ISA, 113  
 master, 195, 197  
 multiplexato, 197  
 PCI, 113-115, 222  
 PCIe, 113-115  
 sincrono, 198-201  
 slave, 195  
 Bus del processore, 581  
 Bus delle periferiche, 581  
 Bus di sistema 184  
 Bus multiplexato, 197  
 Byron, Lord, 15  
 Byte, 76-77, 357  
 Byte di prefisso, 286, 378  
**C**  
 Cache, 83, 314-320  
     a corrispondenza diretta, 316-318  
     di secondo livello, 315  
     protocollo MESI, 609-611  
     separata, 84, 315, 358  
     set-associativa, 319-321  
     snooping, 234, 607  
     strategia di aggiornamento, 609  
     unificata, 85  
     write-allocate, 310, 611  
     write-back, 309, 320  
     write-deferred, 320  
     write-through, 320, 608  
 Cache, miss, 318  
 Cache, strategia d'invalidazione, 609  
 Cache, strategia di aggiornamento, 609  
 Cache, write-back, 320  
 Cache, write-deferred, 320  
 Cache, write-through, 320, 599  
 Cache a blocchi, 492  
 Cache a corrispondenza diretta, 316  
 Cache delle micro-operazioni, 325  
 Cache di secondo livello, 315  
 Cache hit, 318  
 Cache Only Memory Access (COMA), 600, 624-625  
 Cache set-associativa, 319  
 Cache set-associativa a n vie, 319  
 Cache specializzata, 85, 315  
 Cache unificata, 85  
 Call gate, 468  
 Capacitore (condensatore), 117  
 Caricamento, 546-558  
 Caricamento speculativo, 439  
 Catamount, 638  
 CCD (*vedi* Charge-Coupled Device)  
 CC-NUMA, 615  
 CD registrabile, 105-108  
 CD riscrivibile, 108  
 CDC (*vedi* Control Data Corporation)  
 CDC 6600, 14, 21  
 CD-R (*vedi* CD registrabile)  
 CD-ROM (*vedi* Compact Disc-Read Only Memory)  
 CD-ROM multisessione, 107  
 Celeron, 45  
 Cella di memoria, 75  
 Central Processing Unit (CPU), 18, 56-72  
 Centro di calcolo, 24  
 Charge-Coupled Device (CDC), 139  
 Checkerboarding, 461  
 Chiamata al supervisore, 11  
 Cluster, 38, 510  
 Chiamata di sistema, 11, 445, 723-725  
 Chiamata ravvicinata, 722  
 Chiave di un record, 476  
 Chip, 162  
 Chip di memoria, 183-186  
 Chipset, 207, 212  
 Chiusura, 682  
 Cicli di bus, 198  
 Ciclo del percorso dati, 57  
 Ciclo del processore, 706  
 Ciclo di clock, 173  
 Ciclo di prelievo-decodifica-esecuzione, 58, 250  
 Ciclo locale, 133  
 Ciclo virtuoso, 30  
 Cilindro, disk, 90  
 Circuiti integrati (IC), 162  
 Circuiti logici digitali, 162-173  
 Circuiti per l'aritmetica, 169  
 Circuito  
     aritmetico, 169-172  
     reti combinatorie, 163-165

Circuito virtuale, 234  
 CISC (*vedi* Complex Instruction Set Computer)  
 Classificazione di pacchetti, 589  
 Clock, 173-174  
 Clone, 25  
 Cloud computing, 39  
 Cluster computing, 640-645  
 Cluster di Google, 641-645  
 Cluster of Workstations, 602  
 COBOL, 40, 369  
 Coda di messaggi, 512  
 Code page, 143  
 Code point, 143  
 Codice di escape, 378  
 Codice operativo (opcode), 250  
 Codice operativo espandibile, 374-376  
 Codice Reed-Solomon, 89  
 Codici condizione, 346  
 Codici correttori, 78-82  
 Codifica, 8b/10b, 233  
 Codifica dei caratteri, 141-145  
 Codifica hash, 544  
 Coerenza della cache, 616  
 Collegamento, 554-558  
     a tempo del binding, 551-554  
     dinamico, 554-558  
     MULTICS, 463, 554  
     UNIX, 557  
     Windows, 555  
 Collegamento, file, 503  
 Collegamento a festone (daisy chaining), 203  
 Collegamento dinamico, 554-558  
     MULTICS, 554-557  
     UNIX, 557  
     Windows, 557  
 Collegamento esplicito, 557  
 Collegamento implicito, 557  
 Collettore, 152  
 Collettore aperto, 196  
 COLOSSUS, 14, 17-18  
 COMA (*vedi* Cache Only Memory Access computer)  
 COMA semplice, 624  
 Commodity Off The Shelf (COTS), 38

Commutatori crossbar, 611  
 Comutazione a livello, latch, 175  
 Comutazione di pacchetto, 584  
 Comutazione sul fronte, flip-flop, 176  
 Compact Disc-Read Only Memory (CD-ROM), traccia, 107  
 Compact Disc-Read Only Memory (CD-ROM), XA, 107  
 Compagnia telefonica, 132  
 Comparatore, 166-167  
 Compilatore, 7, 526  
 Complemento a due, 688  
 Complemento a uno, 688  
 Complex Instruction Set Computer (CISC), 62  
 Compute Unified Device Architecture (CUDA), 591  
 Computer  
     gioco, 36-37  
     parallelo sui dati, 72-74  
     usa e getta, 32-34  
 Computer, approccio strutturale, 1-13  
 Computer, tipologie, 29-40  
 Computer a valvole, 16-19  
 Computer con parallelismo sui dati, 70-72  
 Computer di generazione zero, 13-16  
 Computer di prima generazione, 16-19  
 Computer di quarta generazione, 24-27  
 Computer di quinta generazione, 27-29  
 Computer di seconda generazione, 19-22  
 Computer di terza generazione, 22-24  
 Computer invisibile, 27  
 Computer MIPS, 62  
 Computer paralleli  
     prestazioni, 657-663  
     tassonomia, 599  
 Computer per giocare, 36  
 Computer usa e getta, 32  
 Comunicatore, 647  
 Condivisione a soglia, 575  
 Condivisione ripartita delle risorse, 574  
 Condivisione totale delle risorse, 575  
 Consistenza  
     debole, 605  
     di cache, 602  
     di processore, 604

dopo rilascio, 605  
 sequenziale, 603  
 stretta, 602  
 Consumatore, 482  
 Contatore di locazioni, 726  
 Control Data Corporation (CDC), 21  
 Controller per videogiochi, 123-125  
 Controllo di flusso, 234  
 Controllore, 111  
 Controllore del disco, 91  
 Conversione tra basi, 684-687  
 Copia dopo scrittura, 497  
 Coprocessore, 582-591  
     grid, 663-666  
     multiprocessore, 595  
     parallelismo nel chip, 562-570  
 Coprogettati, hardware e software, 28  
 Core 2 duo, 44  
 Core i7  
     banking, 340  
     formati d'istruzione, 371  
     foto del chip, 44  
     hyperthreading, 42, 573-575  
     indirizzamento, 392-394  
     introduzione, 41-50  
     istruzioni, 403  
     livello ISA, 366-369  
     memoria virtuale, 464-468  
     microarchitettura, 335-341  
     modalità d'indirizzamento, 392-394  
     modello di memoria, 537  
     multiprocessore, 577  
     pipelining, 213  
     predizione dei salti, 321-326  
     reorder buffer, 338  
     schedulatore, 327  
     tipi di dati, 367  
     unità di ritiro, 340  
     virtualizzazione, 471  
 CoreConnect, 581  
 Coroutine, 415, 421-423  
 Corsa critica, 482-486  
 Corsia, PCI Express, 233  
 Cortex A9, 341-345  
 Costante di rilocazione, 549  
 COTS (*vedi* Commodity Off The Shelf)  
 COW (*vedi* Cluster of Workstations)

CP/M, 24  
 CPU (*vedi* Central Processing Unit)  
 CPU, chip, 191-193  
 Cray, Seymour, 21  
 CRAY-1, 14  
 CRC (*vedi* Cyclic Redundancy Check)  
 Creazione di un processo, 481  
 Crittografia,  
     a chiave pubblica, 594  
     a chiave simmetrica, 594  
 Crittoprocessori, 594  
 CRT (*vedi* Tubo catodico)  
 Cubo, 629  
 CUDA (*vedi* Compute Unified Device Architecture)  
 Cut through virtuale, 634  
 Cyclic Redundancy Check, 233

**D**

Data center, 39  
 Dati scaduti, 607  
 DDR (*vedi* Double Data Rate RAM)  
 DEC (*vedi* Digital Equipment Corporation)  
 DEC Alpha, 14, 26  
 DEC PDP-1, 14, 20, 76  
 DEC PDP-11, 14, 24  
 DEC PDP-8, 14, 20  
 DEC VAX, 14, 61  
 Decodifica dell'indirizzo, 243  
 Decodifica parziale dell'indirizzo, 242  
 Decodificatore, 165-166  
 Demultiplexer, 165  
 Descrittore di file, 502, 724  
 Descrittore di sicurezza, 510  
 Destinatario, 227  
 Determinazione del destinatario, 589  
 Device bus dei registri di periferica, 582  
 Diametro, rete, 627  
 Difference engine (macchina differenziale), 15  
 Digital Equipment Corporation, 14, 20  
 Digital Subscriber Line, 132-134  
 Digital Subscriber Line Access Multiplexer (DSLAM), 134  
 Dimensionalità, 628  
 DIMM (*vedi* Dual Inline Memory Module)

DIP (*vedi* Dual Inline Package)  
 Dipendenza, 307, 329  
 Dipendenza effettiva, 307  
 Dipendenza RAW, 307  
 Dipendenza WAR, 329  
 Direct Memory Access (DMA), 212, 406  
 Directory, 471  
 Directory delle pagine, 466  
 Directory di lavoro, 502  
 Directory radice, 502  
 Direttive dell'assemblatore, 530  
 Disallineamento del bus, 115, 197  
 Disambiguazione, 341  
 Dischi magnetici, 87-99  
 Dischi ottici, 101-105  
 Disco  
   ATAPI, 93  
   CD-ROM, 102-105  
   DVD, 108-110  
   IDE, 92-94  
     magnetico, 88-92  
     ottico, 102-111  
   RAID, 95-99  
   SCSI, 94-95  
   SSD, 97-99  
     Winchester, 90  
 Disco, ATA-3, 93  
 Disco, ATAPI-4, 93  
 Disco, ATAPI-5, 93  
 Disco, Winchester, 90  
 Disco a stato solido (SSD), 99  
 Disco digitale versatile, 109  
 Disco video digitale, 109  
 Dispositivi di Input/Output  
   apparecchiatura per le telecomunicazioni, 130-138  
   controller per videogiochi, 123  
   fotocamera digitale, 139  
   modem, 130  
   mouse, 121  
   schermo piatto, 118  
   stampante a colori, 127  
   stampante a getto d'inchiostro, 128  
   stampante laser, 125  
   stampanti speciali, 129  
   tastiera, 116  
   touch screen, 116

Dispositivo a tre stati, 183  
 Disposizione fisica dei contatti (pinout), 49  
 Distanza di Hamming, 79  
 Divisore, 134  
 DLL (*vedi* Dynamic Link Library)  
 DMA (*vedi* Direct Memory Access)  
 Dot, 726  
 Double Data Rate RAM (DDR), RAM, 188  
 Dpi (punti per pollice), 126  
 DRAM (*vedi* Dynamic RAM)  
 Driver del bus, 195  
 DSL (*vedi* Digital Subscriber Line)  
 DSLAM (*vedi* Digital Subscriber Line Access Multiplexer)  
 DSM (*vedi* Memoria condivisa distribuita)  
   hardware, 615  
 Dual Inline Memory Module (DIMM), 86  
 Dual Inline Package (DIP), 162  
 Duali, 159  
 Dump di memoria, 10  
 DVD (*vedi* disco video digitale)  
 Dynamic Link Library (DLL), 555

**E**

Eckert, J. Presper, 17  
 ECL (*vedi* Emitter-Coupled Logic)  
 EDO (*vedi* Extended Data Output)  
 EDSAC, 14  
 EDVAC, 17  
 EEPROM (*vedi* Electrically Erasable PROM)  
 EHCI (*vedi* Enhanced Host Controller Interface)  
 EIDE (*vedi* Extended IDE)  
 EISA bus (*vedi* Extended ISA bus)  
 Elaborazione di pacchetti, 588  
 Elaborazione in entrata, 589  
 Elaborazione in uscita, 588  
 Elaborazione parallela, 481  
 Electrically Erasable PROM (EEPROM), 189  
 Emettitore, 152  
 Emitter-Coupled Logic (ECL), 154  
 Emulazione, 23  
 Endian Endian  
   big, 77-78  
   little, 77-78

Enhanced Host Controller Interface (EHCI), 238  
 ENIAC, 14, 17  
 ENIGMA, 16  
 EPIC (*vedi* Explicitly Parallel Instruction Computing)  
 Epilogo della procedura, 420  
 EPROM (*vedi* Erasable PROM)  
 EPT (*vedi* Extended Page Table)  
 Equivalenza tra circuiti, 158-162  
 Erasable PROM (EPROM), 188  
 Errore di pagina, 453  
 Errore relativo, 695  
 Esadecimale, 683  
 Esecuzione condizionata, 437  
 Esecuzione dell'istruzione, 58-61  
 Esecuzione fuori sequenza, 326-332  
 Esecuzione speculativa, 332  
 Esponente, 694  
 Estensione del segno, 256  
 Estrazione di campi, 589  
 Estridge, Philip, 25  
 Ethernet, 583  
 Etichetta, 705  
   linguaggio assemblatore, 726  
 Etichetta globale, 729  
 Evento, 517  
 Evoluzione delle macchine multilivello, 8-13  
 Explicitly Parallel Instruction Computing (EPIC), 433  
 Extended Data Output (EDO), 187  
 Extended IDE (EIDE), 93  
 Extended ISA (EISA), 113  
 Extended Page Table (EPT), 473

**F**

Falsa condivisione, 651  
 Famiglie di computer, esempi, 40  
 Fanout, 627  
 Far call, 722  
 Fast Page Mode (FPM), 187  
 FAT (*vedi* File Allocation Table)  
 Fat tree, 629  
 Field Programmable Gate Array (FPGA), 26, 190-191, 586

FIFO, algoritmo (*vedi* Algoritmo First-In First-Out)  
 File, 474-475  
 File Allocation Table (FAT), 507  
 File binario eseguibile, 726  
 File immediato, 512  
 Filtro, 502  
 Filtro di Bayer, 139  
 Firewall, 586  
 Flag della modalità d'incremento, 716  
 Flag di parità, 719  
 Flag di riporto ausiliario, 715  
 Flip-flop, 176  
 Floppy disk, 89  
 Flusso, 415-417  
   coroutine, 421-423  
   diramazioni, 415  
   interrupt, 424-427  
   procedure, 416-421  
   trap, 423  
 Flusso di controllo sequenziale, 415-416  
 Formati d'istruzione, 371-381  
   ARM, 378-380  
   ATmega168, 380  
   Core i7, 377  
      criteri progettuali, 372-374  
 Forrester, Jay, 19  
 FORTRAN, 10, 404  
 FORTRAN Monitor System (FMS), 10  
 Fotocamera digitale, 139-141  
 FPGA (*vedi* Field Programmable Gate Array)  
 FPM (*vedi* Fast Page Mode)  
 Frame, 103  
 Frammentazione, 456  
   esterna, 457  
   interna, 461  
 Frame pointer (FP), 362  
 Frazione, 694  
 Frequency shift keying, 131  
 Frequenza del retino dei mezzitoni, 127  
 Frequenza di fallimento (miss ratio), 85  
 Frequenza di successi (hit ratio), 84  
 Full handshake, 202  
 Full-duplex, 132  
 Funzione booleana, 154-155  
 Furto di cicli, 113, 407

**G**

Gamut (gamma dei colori), 128  
 GDT (*vedi* Global Descriptor Table)  
 General Purpose GPU (GPGPU), 592  
 Generazione del codice, 727  
 Generazione della checksum, 590  
 Gerarchia di memoria, 87  
 Gestione dei processi, 512  
   UNIX, 512  
   Windows 7, 515  
 Gestione dell'intestazione, 590  
 Gestione della coda, 590  
 Gestore dell'interrupt, 112  
 Gestore di trap, 423  
 Gettone di accesso, 509  
 Ghosting, 117  
 Global Descriptor Table (GDT), 464  
 Goldstine, Herman, 18  
 GPGPU (*vedi* General Purpose GPU)  
 GPU (*vedi* Graphics Processing Unit)  
 GPU Fermi, 70-71, 591-594  
 Graphical User Interface (GUI), 25, 493  
 Graphics Processing Unit (GPU), 591  
 Grid computing, 663-666  
 GridPad, 14  
 Gruppi d'istruzioni, Itanium 2, 434  
 GUI (*vedi* Graphical User Interface)

**H**

Half adder (sommatore), 170  
 Half-duplex, 132  
 Hamming, Richard, 31  
 Handle, 496  
 Hardware, 8  
   in relazione al software, 8  
 Harvard Mark I, 16  
 Hashing, 544  
 Hawkins, Jeff, 28  
 Hazard (rischio), 307  
 Headless workstation, 640  
 Hello World; esempio, 740  
 High Sierra, 105  
 Hit ratio, 84  
 Hoagland, Al, 31  
 Hoff, Ted, 41

HTTP (*vedi* HyperText Transfer Protocol), 584  
 Hub principale, 236  
 HyperText Transfer Protocol (HTTP)  
 Hyperthreading, Core i7, 573-576  
 Hypervisor, 472

**I**

I/O (*vedi* Input/Output)  
 I/O mappato in memoria, 241  
 I/O programmato, 404  
 IA-64, 431-439  
 IAS, 14  
 IAS, macchina, 18  
 IBM 360, 14, 22-23  
 IBM 701, 19  
 IBM 704, 19  
 IBM 709, 10, 19  
 IBM 7094, 14, 20-22  
 IBM 801, 62  
 IBM 1401, 14, 20, 21-22  
 IBM CoreConnect, 581  
 IBM Corporation, 19-21  
 IBM PC, 14, 24-25  
 IBM POWER4, 14, 27  
 IBM PS/2, 42  
 IBM RS6000, 14  
 IBM Simon, 14  
 IC (*vedi* Circuiti integrati)  
 Identificatore dello spazio degli indirizzi, 470  
 IFU (*vedi* Unità di prelievo dell'istruzione)  
 JVM, 297  
   codice Java, 273-274  
   implementazione, 279-291  
   insieme d'istruzioni, 268-273  
   modello della memoria, 260-268  
   stack, 264-267  
 ILC (*vedi* Instruction Location Counter)  
 ILLIAC, 17, 70  
 Implementazione, 449-452  
 Implementazione della funzionalità macro, 536  
 Inchiostro  
   a base di coloranti, 129  
   a pigmenti, 129  
   solido, 1296

Indice destinazione, 710  
 Indice sorgente, 710  
 Indirizzamento, 371, 381-396  
   8088, 706-710  
   a registro, 382  
   a registro con indice, 714  
   a registro con indice e spiazzamento, 714  
   a registro indiretto, 382  
   a stack, 386  
 ARM, 394  
 ATmega 219, 394-395  
   con spiazzamento, 714  
 Core i7, 392-394  
   diretto, 382  
   immediato, 381  
   implicito, 715  
   indicizzato, 384-398  
   indicizzato esteso, 385  
   istruzioni di salto, 389  
 Indirizzo, 75  
 Indirizzo di memoria, 75  
 Indirizzo effettivo, 716  
 Indirizzo lineare, 466  
 Industry Standard Architecture (ISA), 113  
 Infisso, 386  
 Iniziatore, bus PCI, 225  
 I-node, 505  
 Input/Output (I/O), 111-146  
 Input/Output, istruzioni, 404-407  
 Insiemi d'istruzioni a confronto, 414  
 Instradamento, 589  
 Instruction Location Counter (ILC), 538  
 Instruction Pointer, 706, 710  
 Instruction Register (IR), 56  
 Integrated drive electronics (IDE), 92  
 Intel 386, 14  
 Intel 4004, 40  
 Intel 8080, 14, 42  
 Intel 8086, 41, 42  
 Intel 8088, 706-710  
   indirizzamento e memoria, 710-715  
   programmi d'esempio, 736-741  
   salto corto, 720  
   salto lungo, 720  
   segmenti, 710  
   insieme d'istruzioni, 715-726  
 Intel 80286, 42  
 Intel 80386, 42  
 Intel 80486, 42  
 Intel Core i7 (*vedi* Core i7)  
 Intel Xeon, 45  
 Interconnessione completa, 629  
 Interconnessione programmabile, 190  
 Interfacce, 239-243  
 Interfaccia a scambio di messaggi (MPI), 646  
 Interfaccia di I/O, 240-243  
 Interfaccia sincrona di memoria, 215  
 Internet Service Provider (ISP), 584  
 Internet via cavo, 134-138  
 Interro double, 697  
 Interpretazione, 2  
 Interprete, 2, 58, 704  
 Interrupt, 112, 424-427  
   impreciso, 330  
   preciso, 330  
   trasparente, 425  
 Introduzione, 726  
 Invalidazione, strategia, 609  
 Invertitore, 152  
 IP, intestazione, 585  
 IP, protocollo, 585  
 Ipercubo, 630  
 IR (*vedi* Instruction Register)  
 Iron Oxide Valley, 31  
 ISA, bus (*vedi* Industry Standard Architecture)  
 ISP (*vedi* Internet Service Provider)  
 Istruzioni  
   ARM 410-412  
   ATmega168, 412-414  
   binarie, 397  
   Core i7, 407-410  
   di ciclo, 403  
   di confronto, 400  
   di Input/Output, 404-407  
   di salto condizionato, 400-402  
   di trasferimento dati, 386  
   livello ISA, 351  
   unarie, 398  
 Istruzioni del linguaggio assemblativo, 528-532  
 Istruzioni di chiamata di procedura, 402  
 Istruzioni di confronto, 400-402

Istruzioni di un byte, 712  
 Istruzioni di una parola, 712  
 Istruzioni per la gestione di directory, 479-480  
 Itanium 2, 431-439  
 Itanium 2, scheduling delle istruzioni, 2, 434

**J**

Java Virtual Machine, 249, 267  
 Jobs, Steve, 25  
 Johnniac, 17  
 Joint Photographic Experts Group (JPEG), 140  
 Joint Test Action Group (JTAG), 213  
 JPEG (*vedi* Joint Photographic Experts Group)  
 JTAG (*vedi* Joint Test Action Group)

**K**

Kilby, Jack, 22  
 Kildall, Gary, 24  
 Kinect, 123

**L**

LAN (*vedi* Local Area Network)  
 Land Grid Array (LGA), 162  
 Land, 110  
 Larghezza di banda complessiva, 658  
 Larghezza di banda del processore, 67  
 Larghezza di banda di bisezione, 627  
 Latch, 175-177  
 Latch D, 175  
 Latch D temporizzato, 170-171, 177  
 Latch SR, 175  
 Latenza rotazionale, 90  
 Latin-1, 143  
 LBA (*vedi* Logical Block Addressing)  
 LCD (*vedi* Schermo a cristalli liquidi)  
 LDT (*vedi* Local Descriptor Table)  
 LED (*vedi* Light Emitting Diode)  
 Legge di Amdahl, 660  
 Legge di De Morgan, 159  
 Legge di Moore, 29  
 Legge di Nathan, 30  
 Leibniz, Gottfried Wilhelm von, 13  
 Letterali, 540

LGA (*vedi* Land Grid Array)  
 Libreria condivisa, 557  
 Libreria destinazione, 558  
 Libreria importata, 557  
 Libreria statica, 558  
 Libro arancione, 107  
 Libro giallo, 102  
 Libro rosso, 101  
 Libro verde, 105  
 Light Emitting Diode (LED), 122  
 Limitazione termica (Thermal throttling), 213  
 Linda, 652  
 Linea di cache, 85, 316  
 Lines Per Inch (LPI), 127  
 Linguaggi ad alto livello, 7  
 Linguaggio, 1  
 Linguaggio assemblativo, 526-529, 704-706  
 Linguaggio assemblativo, istruzioni, 528  
 Linguaggio destinazione, 525  
 Linguaggio macchina, 1, 704  
 Linguaggio sorgente, 525  
 Linkage editor, 546  
 Linker, 547, 726  
 compiti, 547-549  
 Linking loader, 546  
 Lisa, 14  
 Lista delle locazioni libere, 478  
 Little endian, memoria, 77  
 Livello, 3  
 dei dispositivi, 4  
 del linguaggio assemblativo, 525-558  
 di microarchitettura, 6, 249-350  
 ISA, 6, 353-440  
 logico digitale, 151-244  
 Livello applicativo, 649, 665  
 Livello dei dispositivi, 4  
 Livello del linguaggio assemblativo, 527-558  
 Livello di architettura dell'insieme d'istruzioni, 353-440  
 Livello di astrazione hardware, 495  
 Livello di microarchitettura, 6, 249-350  
 esempi, 334-341  
 progettazione, 291-301  
 Livello di raccolta, 664

Livello di transazione, PCI Express, 234

Livello ISA, 6, 355  
 ARM, 363-365  
 ATmega168, 366-367  
 Core i7, 361-363  
 formati d'istruzione, 371-381  
 indirizzamento, 381-396  
 panoramica, 355-367  
 tipi d'istruzioni, 396-414  
 tipi di dati, 367-371  
 Livello ISA, proprietà, 355  
 Livello logico digitale, 5  
 bus, 194-196, 214-232  
 bus PCI, 215-223  
 bus PCI express, 231-235  
 chip della CPU, 192-193  
 circuiti, 162-172  
 interfacce di I/O, 239-240  
 memoria, 169-185  
 porte logiche, 152-162  
 Livello macchina del sistema operativo, 6, 437-510  
 Livello risorse, 664  
 Livello software, 234  
 Livello struttura, 664  
 Local Descriptor Table (LDT), 464  
 Local Area Network (LAN), 583  
 Località spaziale, 316  
 Località temporale, 316  
 Locazione di memoria, 75  
 Logica negativa, 161  
 Logica positiva, 161  
 Logical Block Addressing (LBA), 93  
 Long, 708  
 Lovelace, Ada, 15  
 LPI (*vedi* Lines Per Inch)  
 LRU, algoritmo (*vedi* Algoritmo Least Recently Used)  
 Lunghezza del percorso, 292  
 LUT, 190

**M**

M.I.T.  
 TX-0, 20  
 TX-2, 20

Macchina del sistema operativo, 7, 445-518  
 Macchina di Von Neumann, 18

Macchina fotografica digitale, 139-141  
 Macchina virtuale, 2, 472  
 Macchine multilivello, 4-12  
 evoluzione, 8  
 Macintosh, Apple, 25  
 Macro, chiamata, 534  
 Macro, con parametri, 535  
 Macro, definizione, 533  
 Macro, espansione, 534  
 Macro, linguaggio assemblativo, 532-537  
 Macro del sistema operativo, 11  
 Macroarchitettura, 254  
 Mainframe, 40  
 MAL (*vedi* Micro Assembly Language)  
 MANIAC, 17  
 Mantissa, 694  
 Mappa di memoria, 449  
 MAR (*vedi* Memory Address Register)  
 Mark I, 14  
 Maschera, 387  
 BlueGene, 631-635  
 Red Storm, 635-638  
 Massively Parallel Processor, 602  
 Master File Table (MFT), 510  
 Masuoka, Fujio, 99  
 Mauchley, John, 16  
 MDR (*vedi* Memory Data Register)  
 Memoria, 74-87, 174-191  
 associativa, 463, 544  
 big endian, 77  
 cache, 82-85  
 CD-ROM, 101, 103  
 COMA, 623  
 d'attrazione, 624  
 flash, 189  
 FPM, 187  
 interlacciata, 614  
 little endian, 77  
 non volatile, 188  
 RAM, 186-191  
 secondaria, 87-111  
 virtuale, 446-471  
 Memoria, condivisa a livello applicativo, 649-657  
 Memoria condivisa distribuita, 598, 650-652  
 Memoria d'attrazione, 624

Memoria e indirizzamento, 8088, 710-715  
 Memoria secondaria, 87-111  
 Memoria virtuale, 447-471  
 ARM, 468-471  
 Core i7, 464-468  
 e caching, 471  
 UNIX, 496-497  
 Windows 7, 498-500  
 Memorie di controllo, 61, 259  
 Memory Address Register (MAR), 255  
 Memory Data Register (MDR), 255  
 Memory Management Unit (MMU), 451  
 Mescolamento perfetto, 613  
 Mesh, 629  
 MESI, protocollo, 610  
 Metal Oxide Semiconductor (MOS), 154  
 Metodo, 402  
 Mezzitoni, 127  
 MFT (*vedi* Master File Table)  
 Mic-1, 259-264, 279  
 Mic-1, notazione 1, 274-279  
 Mic-2, 301  
 Mic-3, 305  
 Mic-4, 310  
 Mickey, 122  
 Micro Assembly Language (MAL), 275  
 Microarchitettura  
 a tre bus, 296  
 ARM, 341-345  
 ATmega168, 346-348  
 Core i7, 338-341  
 Mic-1, 274-291  
 Mic-2, 301-304  
 Mic-3, 305-310  
 Mic-4, 310-313  
 OMAP 4430, 341-345  
 Microarchitettura della CPU Core i7, 335  
 Microcodice, 11-13  
 Microcontrollore, 34  
 Microdrive, 140  
 MicroInstruction Register (MIR), 261  
 Microistruzione, 61, 257  
 Microprogram counter (MPC), 261  
 Microprogramma, 6, 707  
 Microprogrammazione, 8, 12  
 Microsoft assembler (MASM), 528

Miglioramento delle prestazioni, 661-663  
 MIMD (*vedi* Multiple Instruction-stream  
 Multiple Data-stream)  
 Minislot, 137  
 MIPS, 14  
 MIR (*vedi* MicroInstruction Register)  
 MMU (*vedi* Memory Management Unit)  
 MMX (*vedi* MultiMedia eXtensions)  
 Modalità di indirizzamento, 381  
 analisi, 395-396  
 ARM, 394, 395  
 istruzioni di salto, 389-390  
 Modalità kernel, 357  
 Modalità reale, 361  
 Modalità utente, 357  
 Modalità virtuale 8086, 362  
 Modello di consistenza, 602  
 Modello di memoria, 357, 731  
 Core i7, 358  
 grande, 731  
 minuscolo, 731  
 piccolo, 731  
 Modem, 130  
 Modulazione, 131  
 Modulazione a coppia di bit, 132  
 Modulazione d'ampiezza, 131  
 Modulazione di fase, 131  
 Modulazione di frequenza, 131  
 Modulo e segno, 688  
 Moore, Gordon, 29  
 MOS (*vedi* Metal Oxide Semiconductor)  
 Motif, 493  
 Motion Picture Experts Group (MPEG),  
 579  
 Motore di elaborazione di pacchetti (PPE),  
 587  
 Motore di elaborazione programmabile,  
 589  
 Motore di stampa, 126  
 Motorola 68000, 61  
 Mouse, 121-123  
 MPC (*vedi* MicroProgram Counter)  
 MPEG-2, 579  
 MPI (*vedi* Interfaccia a scambio di messaggi)  
 MPP (*vedi* Massively Parallel Processor)  
 MS-DOS, 26

Multicomputer, 73, 596-602, 625, 645-648  
 BlueGene, 631-635  
 cluster di Google, 641-645  
 MPP, 602  
 Red Storm, 635  
 Multicomputer, prestazioni, 657-663  
 Multicomputer, software, 645, 658  
 Multicomputer a scambio di messaggi,  
 625-657  
 MULTICS (*vedi* MULTplexed Information  
 and Computing Service)  
 MultiMedia eXtensions (MMX), 43  
 Multiple Instruction-stream Multiple Data-  
 stream (MIMD), 600  
 MULTplexed Information and Computing  
 Service (MULTICS), 463, 554  
 Multiplexer, 164  
 Multiprocessore, 72, 595-599  
 COMA, 623-625  
 Core i7, 577-578  
 e multicomputer, 594-602  
 eterogeneo, 578-582  
 NUMA, 600, 614-623  
 simmetrico, 606-614  
 Multiprocessore basato su directory, 617  
 Multiprocessori eterogenei, 578-582  
 Multiprocessori in un solo chip, 576-582  
 Multiprocessori omogenei, 577-578  
 Multiprogrammazione, 23  
 Multithreading a grana fine, 570  
 Multithreading a grana grossa, 571  
 Multithreading nel chip, 570-576  
 Multithreading simultaneo, 209, 572  
 Mutex, 514  
 Mutua capacità, 117  
 Myhrvold, Nathan, 30

## N

NaN (*vedi* Not a Number)  
 NC-NUMA (*vedi* No Cache NUMA)  
 NEON, 343  
 Network Interface Device (NID), 134  
 Network of Workstations, 602  
 Newton, 14, 47  
 Nibble, 408  
 NID (*vedi* Network Interface Device)

No Cache NUMA (NC-NUMA), 615  
 No Remote Memory Access (NORMA), 602  
 Nome simbolico, 704  
 Nomi mnemonici, 704, 726  
 NonUniform Memory Access (NUMA), 600,  
 615  
 NORMA (*vedi* No Remote Memory Access)  
 Not a Number (NaN), 700  
 Notazione in eccesso, 688  
 Notazione polacca, 386  
 Notazione polacca inversa, 386  
 Notazione postfissa, 386  
 NOW (*vedi* Network of Workstations)  
 Noyce, Robert, 22  
 NT file system (NTFS), 507  
 NTFS (*vedi* NT file system)  
 NTOS, livello esecutivo, 495  
 NUMA, 614-623 (*vedi* Nonuniform Memory  
 Access)  
 Numeri binari  
 conversione tra basi, 684-687  
 negativi, 688-689  
 somma, 690  
 Numero a precisione doppia, 368  
 Numero a precisione finita, 681  
 Numero decimale, 683  
 Numero denormalizzato, 696  
 Numero esadecimale, 728  
 Numero in virgola mobile, 695-696  
 Numero in virgola mobile, normalizzato,  
 696  
 Numero ottale, 728  
 Nvidia Fermi GPU, 591-594  
 Nvidia Tegra, 36, 47

## O

OC-IP (*vedi* Open Core Protocol-  
 International Partnership)  
 Oggetto, file, 726  
 OGSA (*vedi* Open Grid Services Architecture)  
 OHCI (*vedi* Open Host Controller Interface)  
 OLED (*vedi* Organic Light Emitting Diode)  
 Olsen, Kenneth, 20  
 OMAP 4430, 342  
 bi-endian, 363  
 cache dati, 343

formati d'istruzione, 378-380  
indirizzamento, 394  
tipi di dati, 370  
Omnibus, PDP-8, 20  
Open Core Protocol-International Partnership (OCP-IP), 582  
Open Grid Services Architecture (OGSA), 666  
Open Host Controller Interface (OHCI), 238  
Operando destinazione, 721  
Operando immediato, 381  
Operando sorgente, 708  
Operazioni binarie, 397-398  
Operazioni unarie, 398-400  
Operazioni del bus, 205-208  
Operazioni della memoria, 255  
Opposto, 400  
Orca, 654-656  
OR-cablunga, 196  
Ordinamento dei byte, 76-78  
Organic Light Emitting Diode (OLED), 120  
Organizzazione della CPU, 56  
Organizzazione della memoria, 180-183  
8088, 706-710  
Organizzazione virtuale, 663  
OS/2, 26  
Osborne-1, 14  
Ottale, 683  
8088, insieme d'istruzioni, 705-726  
Ottetto, 76, 357  
Overlay, 447

**P**

pacchetti d'istruzioni (bundle), 435  
caricamenti speculativi, 429  
EPIC, modello, 433  
scheduling delle istruzioni, 434-436  
Pacchetto, 581, 584, 618  
PCI, 222 PCI  
Pacchetto di acknowledgment, 233  
Pagina, 448  
Pagina gemella, 651  
Pagina impegnata (committed), 498  
Pagina libera, 498  
Pagina riservata, 498  
Paginazione, 447-449

Paginazione a richiesta, 452-456  
Palm PDA, 28  
Parallel Input/Output (PIO), 239  
Parallel Virtual Machine (PVM), 646  
Parallelismo a livello d'istruzione, 65-69, 563-570  
Parallelismo a livello di processore, 69-73  
Parallelismo, nel chip, 562-570  
Parametri attuali, 535  
Parametri delle macro, 535  
Parametri di valutazione software, 658  
Parametri formali, 535  
Parola, 76  
Parola di codice (codeword), 79  
Pascal, Blaise, 13  
Payload, pacchetto PCI, 237, 585  
PBGA (ball grid array), 218  
PC (*vedi* Program Counter)  
PCI, bus (*vedi* Peripheral Component Interconnect, bus)  
PCI Express, 230-235  
architettura, 231-239  
PCI Express, bus (PCIe, bus), 113  
PCI Express, livello fisico, 233  
PCI Express, pila di protocolli, 232  
PCIe (*vedi* PCI Express)  
PCIe, bus (*vedi* PCI Express, bus)  
PDA (see Personal Digital Assistant/PDA (*vedi* Personal Digital Assistant))  
PDP-1, 14, 20  
PDP-11, 14, 24  
PDP-8, 14, 20  
Pentium 4, 26, 42, 43  
Percorso assoluto, 502  
Percorso dati, 6, 56-57, 250-260  
Mic-1, 259  
Mic-2, 301  
Mic-3, 305  
Mic-4, 310  
Percorso dati, temporizzazione, 253-255  
Percorso relativo, 503  
Peripheral Component Interconnect (PCI), bus, 113, 222  
arbitraggio, 226  
segnali, 227  
transazioni, 229

Personal computer, 24-27, 37  
Personal Digital Assistant (PDA), 28  
Pervasive computing (computazione pervasiva), 29  
PGA (*vedi* Pin Grid Array)  
Pietre miliari nell'architettura dei computer, 13-28  
Pin Grid Array (PGA), 162  
PIO (*vedi* Parallel Input/Output)  
Pipe, 236, 512  
Pipeline, 65, 305  
a sette stadi, 310  
Mic-3, 305  
Mic-4, 310  
OMAP, 341  
Pipeline, memoria del Core i7, 213  
Pipeline, stadi, 65  
Pipeline in stallo, 308  
Pipeline U, 67  
Pipeline V, 67  
Pit, 102  
Pixel, 120  
Plain Old Telephone Service (POTS), 133  
PlayStation 3, 36  
Poison bit, 334  
Politica di sostituzione delle pagine, 454-456  
Politica write-allocate, 609  
Porta, 5, 152-154  
Portable Operating System-IX (POSIX), 491  
Portante, 130  
Porzione costante di memoria, 267-270  
POSIX (*vedi* Portable Operating System-IX)  
Posizione di ritardo, 322  
POTS (*vedi* Plain Old Telephone Service)  
Power gating, 216  
PPE (*vedi* Motore di elaborazione di pacchetti)  
Preamble, 89  
Predicati, 436-438  
Predittore dei salti, Core i7, 338  
Predizione dei salti, 321-326  
dinamica, 312-315, 323-326  
statica, 321  
Prefetching, 65, 662  
Prefisso, 410

Prestazioni  
della rete, miglioramento, 590  
miglioramento, 661  
parametri di valutazione hardware, 657  
parametri di valutazione software, 658  
Prestazioni, miglioramento, 661  
Prestazioni dei computer paralleli, 657-663  
Principi di progettazione, 63-65  
Princípio di località, 84, 454  
Problema dei riferimenti in avanti, 537  
Problema della rilocazione, 549  
Procedura, 402, 416-421  
Procedura ricorsiva, 416  
Processo di assemblaggio, 537-544  
Processo figlio, 512  
Processo genitore, 512  
Processore di rete, 583-594  
Processore grafico, 591-594  
Processore, 55-74  
Processore, bi-endian, 363  
Processore vettoriale, 72  
Processori di rete, incremento delle prestazioni, 590  
Progetto giapponese di quinta generazione, 27  
Program Counter (PC), 56, 364, 706  
Program Status Word (PSW), 360, 467  
Programma, 1  
Programma binario, 704  
Programma eseguibile binario, 526, 546  
Programma indipendente dalla posizione, 553  
Programmabile ROM, 188  
Programmatore di sistema, 7  
Programmazione in linguaggio assemblativo, 726  
Prologo della procedura, 420  
PROM (*vedi* Programmable ROM)  
Protocollo, 584  
bus, 191  
IP, 585  
MESI, 606  
PCI Express, 230-235  
Protocollo di coerenza delle cache, 607  
Protocollo internet, 584  
Protocollo write-back, 609  
Protocollo write-once, 609  
Pseudoistruzione, 530-532, 692, 718

PSW (*vedi* Program Status Word)  
 Pthread, 514  
 Puntatore, 382  
 Puntatore allo stack, 709  
 Puntatore base, 709  
 Puntatore di lettura/scrittura, 724  
 Punto d'ingresso, 550  
 PVM (*vedi* Parallel Virtual Machine)

**R**

Radio Frequency IDentification (RFID), 32  
 RAID (*vedi* Redundant Array of Inexpensive Disks)

RAM (*vedi* Random Access Memory)  
 RAM della scheda video, 120-121  
 RAM dinamica, 187  
 RAM statica (SRAM), 187

Random Access Memory (RAM), 74-87,  
 186-191  
 DDR, 188  
 RAM dinamica, 187  
 SDRAM, 187

Ranging, 137

Read Only Memory (ROM), 188

Record d'attivazione, 362, 709

Record logico, 475

Red Storm, 635-640

Reduced Instruction set Computer (RISC),  
 62  
 e CISC, 62-63

principi di progettazione, 63-65

Redundant Array of Inexpensive Disks  
 (RAID), 96

Refresh (ricarica), 187

Registrazione perpendicolare, 90

Registri di segmento, 710

Registri, 5, 180, 360, 708  
 di flag, 360  
 PSW, 360

Registro, di una parola, 712

Registro, indice, 709

Registro a scorrimento, 169

Registro a scorrimento della storia dei salti,  
 326

Registro base, 708

Registro contatore, 708

Registro dati, 708  
 Registro dei codici di condizione, 716  
 Registro di flag, 360 710  
 Registro di un byte, 712  
 Registro generale, 708-708  
 Registro H, 293  
 Registro puntatore, 709-710  
 Registro vettoriale, 72  
 Registro virtuale, 264  
 ReOrder buffer (ROB), Core i7, 338  
 Replicated worker model, 653  
 Rete  
 anello, 577  
 Ethernet, 583  
 LAN, 583  
 store-and-forward, 584  
 wan, 583  
 Rete a commutazione multilivello, 612-614  
 Rete bloccante, 614  
 Rete non bloccante, 612  
 Rete omega, 613  
 Rete store-and-forward, 584  
 Reti, introduzione, 586-588  
 Reti combinatorie, 163-168  
 Reti d'interconnessione, 626-630  
 larghezza di banda di bisezione, 627  
 topologia, 627  
 Retrocompatibile, 354  
 RFID (*vedi* Radio Frequency IDentification)  
 Ricevitore del bus, 195  
 Ricorsione, 402  
 Riduzione del percorso, 294-301  
 Riferimento esterno, 549  
 Rinomina dei registri, 326-332  
 RISC (*vedi* Reduced Instruction Set Computer)  
 RISC e CISC, 62-63  
 Ritardo della porta, 163  
 ROB (*vedi* ReOrder Buffer)  
 ROM (*vedi* Read Only Memory)  
 Route Instradamento, 589  
 Router, 583

**S**

Salto condizionato, 720-721  
 Sandy Bridge, 208, 335

Scala dinamica di tensione, 217  
 Scale, Index, Base byte (SIB), 378, 393  
 Scambio di messaggi bufferizzato, 646  
 Scambio di messaggi non bloccante, 646  
 Scambio sincrono di messaggi, 646  
 Scanner delle pagine, 616  
 Scheda madre, 111  
 Schedulatore, Core i7, 337  
 Scheduling, multicompiler, 648-649  
 Schermi a matrice passiva, 119  
 Schermo, 118  
 Schermo a cristalli liquidi (LCD), 118  
 Schermo a matrice attiva, 120  
 Schermo multitouch, 117  
 Schermo piatto, 118  
 Schermo TFT, 120  
 Scoreboard, 328  
 SCSI (*vedi* Small Computer System Interface)  
 SDRAM (*vedi* Synchronous DRAM)  
 Seastar, 636  
 Secchielli (bucket), 545  
 Seconda passata, assemblatore, 727  
 Security ID (SID), 509  
 Seek (ricerca), 90  
 Segment override, 725  
 Segmentazione, 457-471  
 implementazione, 461  
 Segmento, 458, 710  
 8088, 706  
 Segmento dati, 711, 720  
 Segmento di codice, 706  
 Segmento di collegamento, 554  
 Segmento extra, 720  
 Segnale assertivo, 185  
 Segnale di controllo, 255  
 Segnale negativo, 185  
 Selezione del percorso, 589  
 Semaforo, 487  
 Semanticà della memoria, 602-606  
 consistenza debole, 604  
 consistenza della cache, 607  
 consistenza di processore, 604  
 consistenza dopo rilascio, 605  
 consistenza sequenziale, 603  
 consistenza stretta, 602  
 Sequenzializzatore, 259

