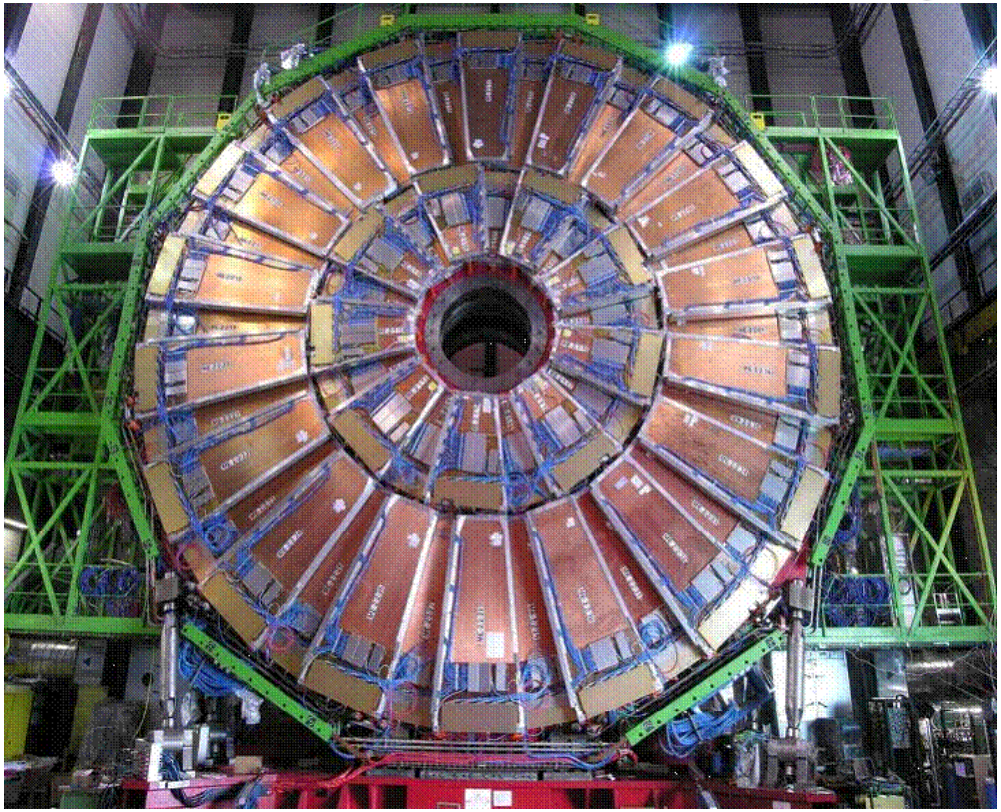
Cathode Strip Chamber (CSC) Raw Data Unpacking and Packing using bit field data classes

Alex Tumanov, Rick Wilkinson

September 2007

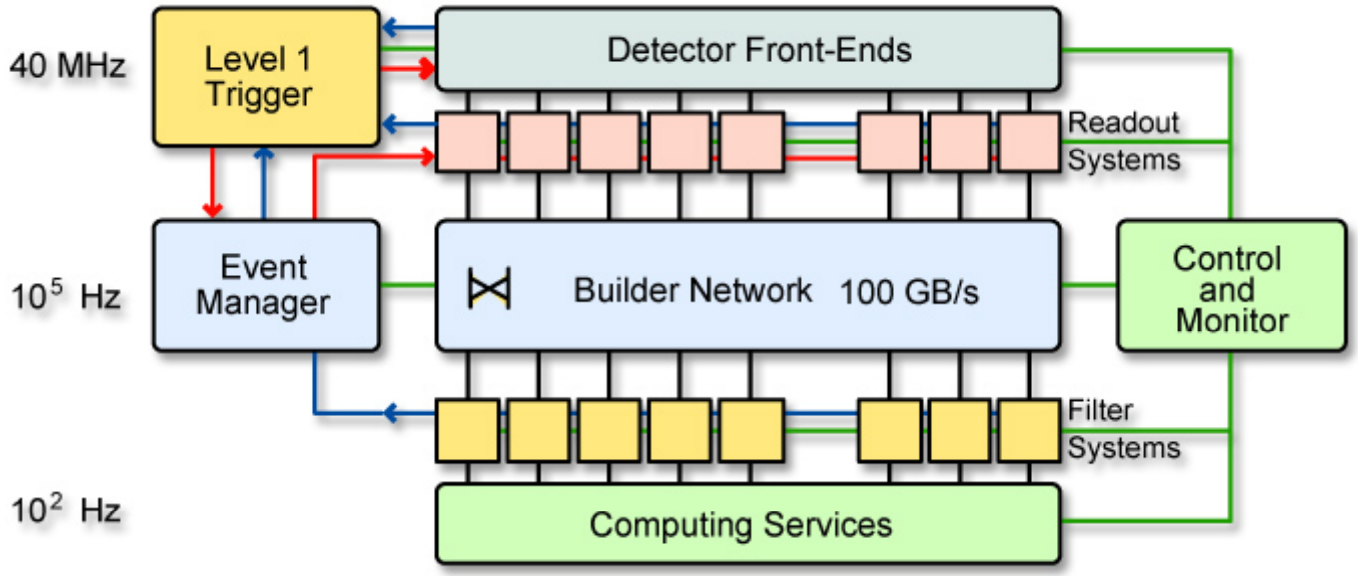
CMS Muon Endcap

- 468 Cathode Strip Chambers send data to Front End Drives (FEDs)
- 8 FEDs with the capacity of 200 MByte/s each
- A typical CSC event size is 10-100Kb
- The L1 event rate is therefore 10-100KHz

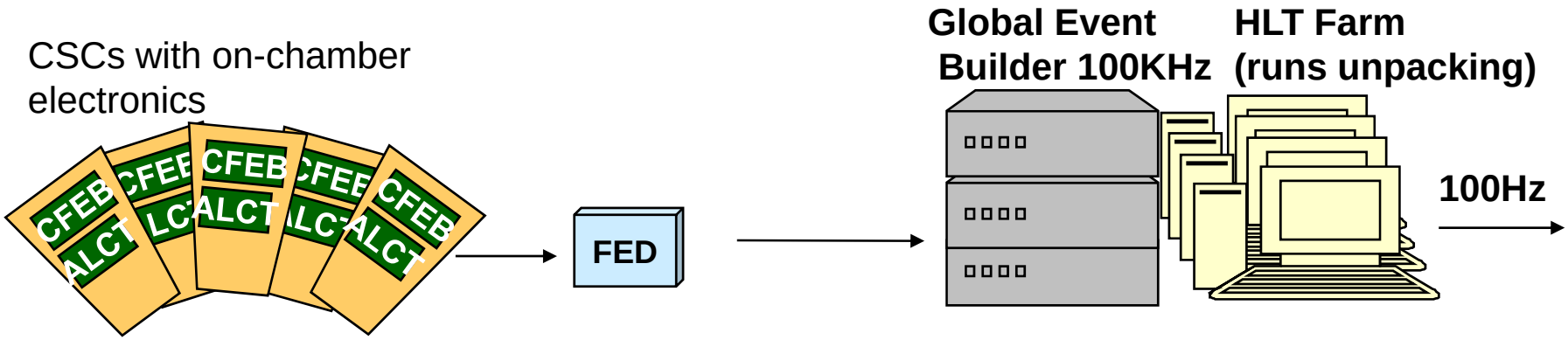


CMS HLT trigger basics

- Every 25ns 20 pp collisions occur
- 800 million pp collisions per second
- L1 event rate of 100KHz
- The CMS HLT trigger which is software based and its output is 100Hz



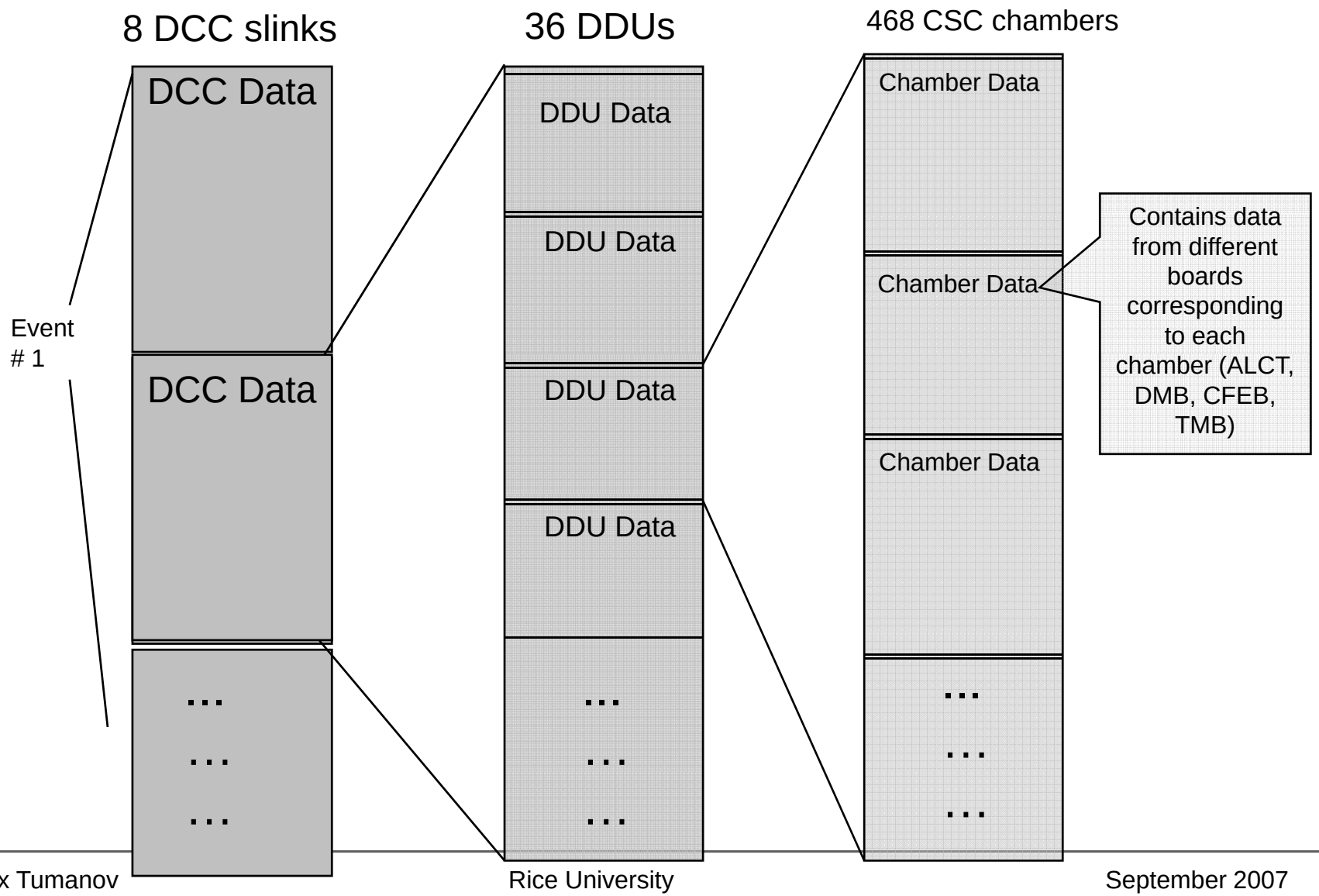
CSC DAQ schematic



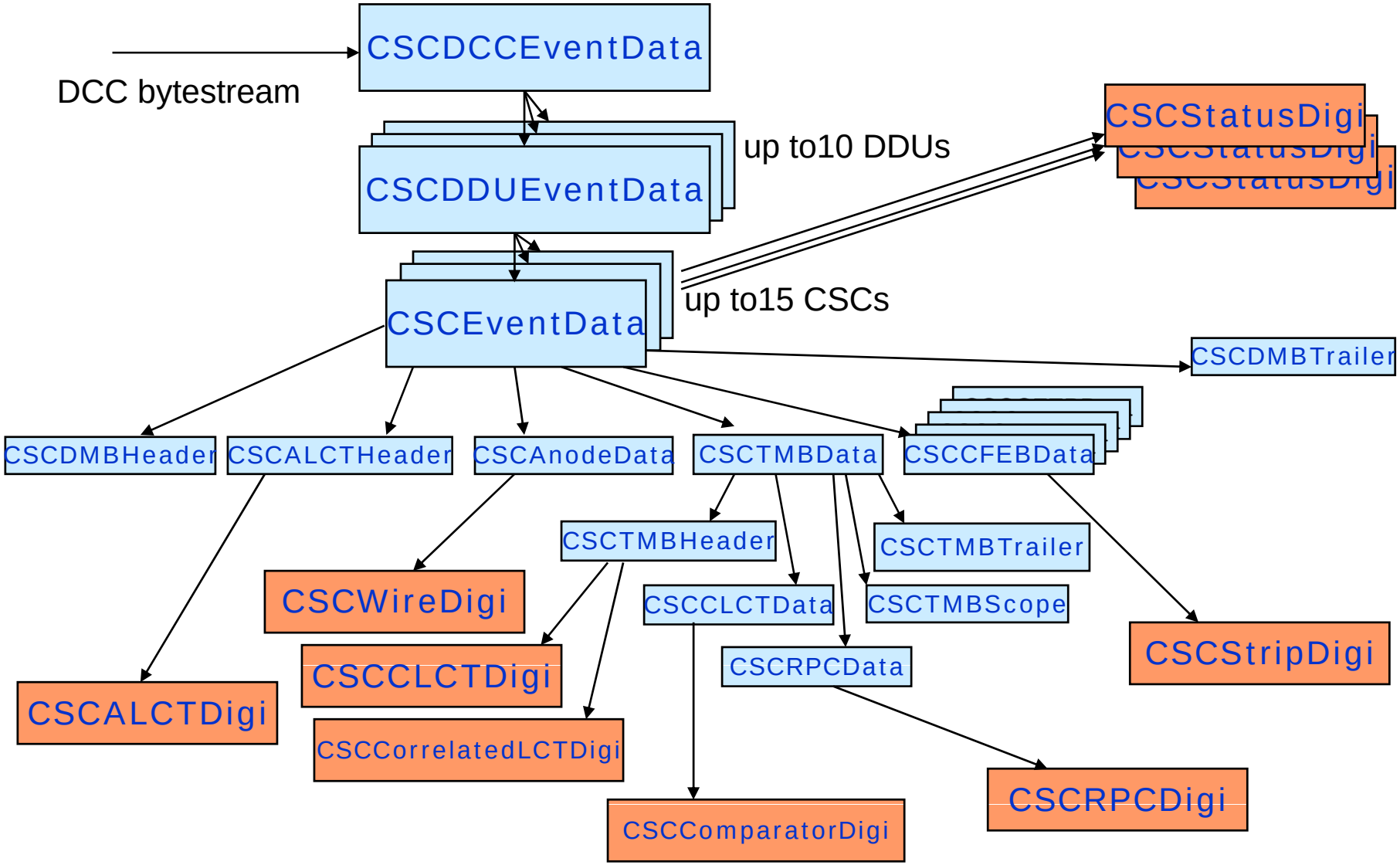
- **CSC Raw data unpacking is a part of the HLT:**
  - ➔ **Runs online**
  - ➔ **Must be very fast in order to handle 100KHz->100Hz**

- **CSC unpacking is also used in offline Reconstruction and local "spy" data acquisition**

CSC Event Data Format

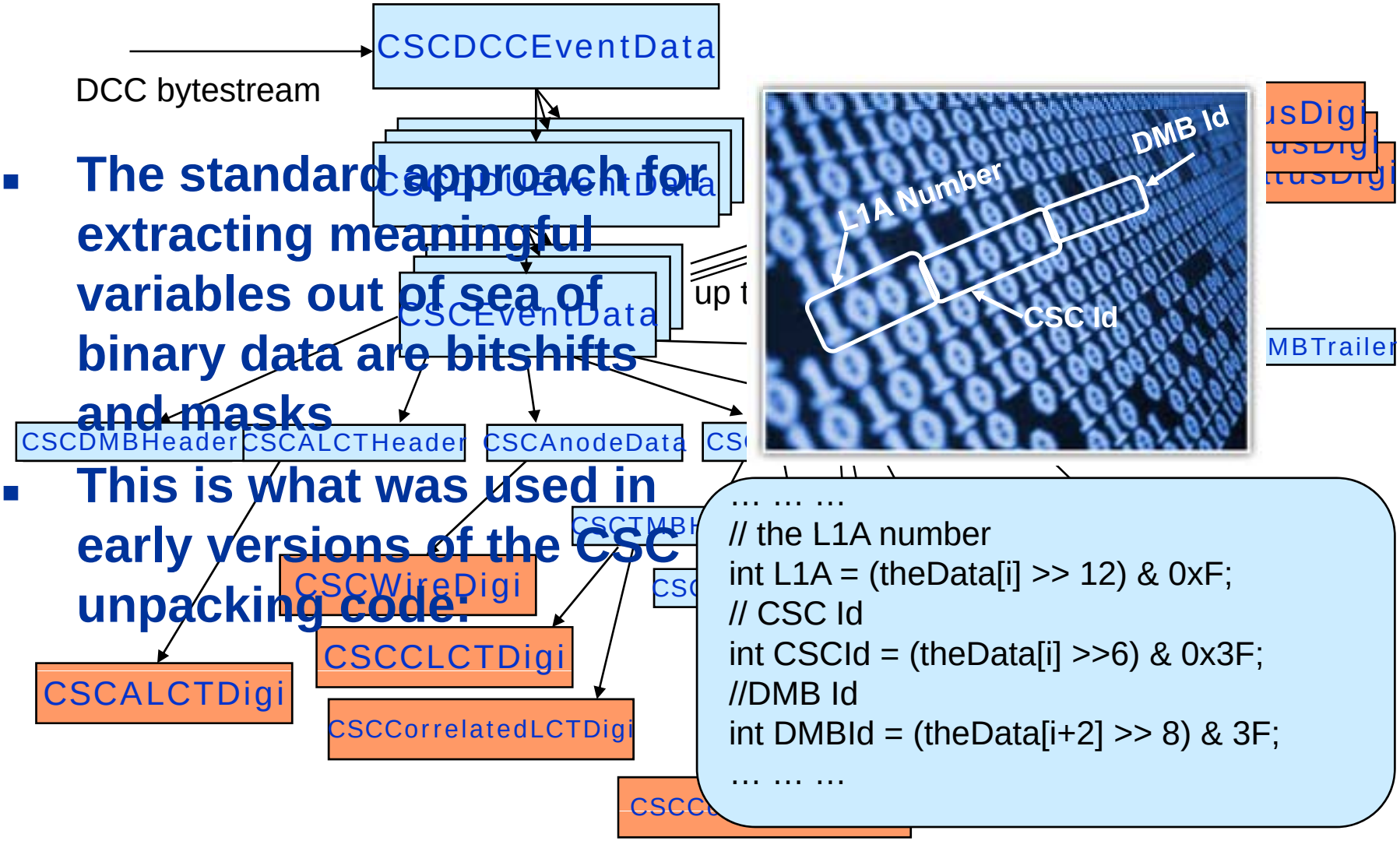


Unpacking classes



Taking a closer look at one class

- **The standard approach for extracting meaningful variables out of sea of binary data are bitshifts and masks**
- **This is what was used in early versions of the CSC unpacking code:**



Binary data formats can be complicated:

- **For LCTxL1A match on CSC:**
  1. **2 Header Words** (all are 'DDU Codes')

Header Word	Highest 4 bits	DDU Code	Lowest 12 bits [11:0]
1a	1001	9	TMB_DAV(1) + ALCT_DAV(1) + CFEB_ACTIVE(5:1) + CFEB_DAV(5:1)
1b	1001	9	DMB_L1A[11:0]
			DMB_L1A[23:12]
			DMB_BXN[11:0]
			TMB_DAV(1) + ALCT_DAV(1) + Active-DAV Mismatch(1) + B_DAV(1) + ALCT_DAV(1) + TMB_DAV(1) + ALCT_DAV(1) + CFEB_DAV(5:1) *recent change, 12/9/02*
			DMB_CRATE(8) + DMB_ID(4)
2			CFEB_MOVLP(5:1) + DMB_BXN[6:0] *recent change, 11/4/02*
2d	1010	A	DMB-CFEB-Sync[3:0] + DMB_L1A[7:0] *recent change, 1/16/03*

  2. **ALCT Data** (if present*) - See Trigger Data Format, below
  3. **TMB Data** (if present*) - See Trigger Data Format, below
  4. **CFEB Data** (CFEB 1 to 5, if present*) - See CFEB Data Format, below
  5. **2 Trailer Words** (both are 'DDU Codes')

Trailer Word	Highest 4 bits	DDU Code	Lowest 12 bits [11:0]
1a	1111	F	Repeat 1a
1b	1111	F	CFEB_MOVLP(5:1) + TMB_HALF(1) + OVLP_HALF(1) + CFEB_HALF(5:1)
1c	1111	F	DMB_L1PIPE(8) + DMB_BXN[3:0]
1d	1111	F	Repeat Header 2b
2a	1110	E	TMB_FULL(1) + CFEB_FULL(5:1) + TMB_MT(1) + CFEB_MT(5:1)
2b	1110	E	ALCT_TIMEOUT(1) + CFEB_ENDTIMEOUT(5:1) + TMB_TIMEOUT(1) + CFEB_STARTTIMEOUT(5:1) *recent change, 12/9/02*
2c	1110	E	Repeat Trailer 2b
2d	1110	E	Repeat Trailer 2b

The First Trailer is a flag for the DDU indicating End Of Event. The Second Trailer is actually the Last Word of the event.

These are our simpler data objects

There are 29 variables in this class

We have 5 (and counting) different versions of the data formats

Better way to unpack data

- **We use bit field classes:**

```
class CSCDMBHeader  {
    public:
      ... ... ...
   unsigned L1ANumber() const {return L1A_;}
   unsigned CSCId() const {return cscId_;}
   unsigned DMBId() const {return dmbId_;}
      ... ... ...
    private:
      ... ... . ..
unsigned L1A_       : 4;
unsigned cfebId_    : 6;
unsigned dmbId_     : 6;
     ... ... ...
};
```

The example of the data class

This is how it gets unpacked

```
CSCDMBHeader(unsigned short * theData) {
   memcpy(this, buf, sizeInWords()*2);
}
```

Old way of unpacking binary data

Binary Data

$$\underbrace{1011}_{\text{L1A}}\underbrace{101110}_{\text{CSC Id}}\underbrace{001011}_{\text{DMB Id}}$$

Fill variables one by one

L1A Number =

CSCId =

DMBId =

Bit Field based unpacking

Binary Data

1011101110 001011

Fill variables at once!!!

L1A Number = $\underbrace{\quad}_{\text{L1A}}$

CSCId = $\underbrace{\quad}_{\text{CSC Id}}$

DMBId = $\underbrace{\quad}_{\text{DMB Id}}$

Why bit field unpacking is faster

- **CSCDMBHeader class contains 29 variables that are filled from binary data**
- **Using bitshifts & masks that would result in 29 lines of code like this:**

```
int L1A = (theData[i] >> 12) & 0xF;
```

- **Using bit field based classes all 29 variables are filled during just one line:**

```
CSCDMBHeader(unsigned short * theData) {memcpy(this, theData,
                                                sizeInWords()*2); }
```

- **Because of this we are able to achieve an order of magnitude of improvement in the average unpacking time (currently around 3ms/event)**
- **Total number of bit field variables in our package – 355**

Understanding memcpy()

- **Memcpy implementation in assember loads two addresses and the size into registers and does the copying in a single hardware instruction**

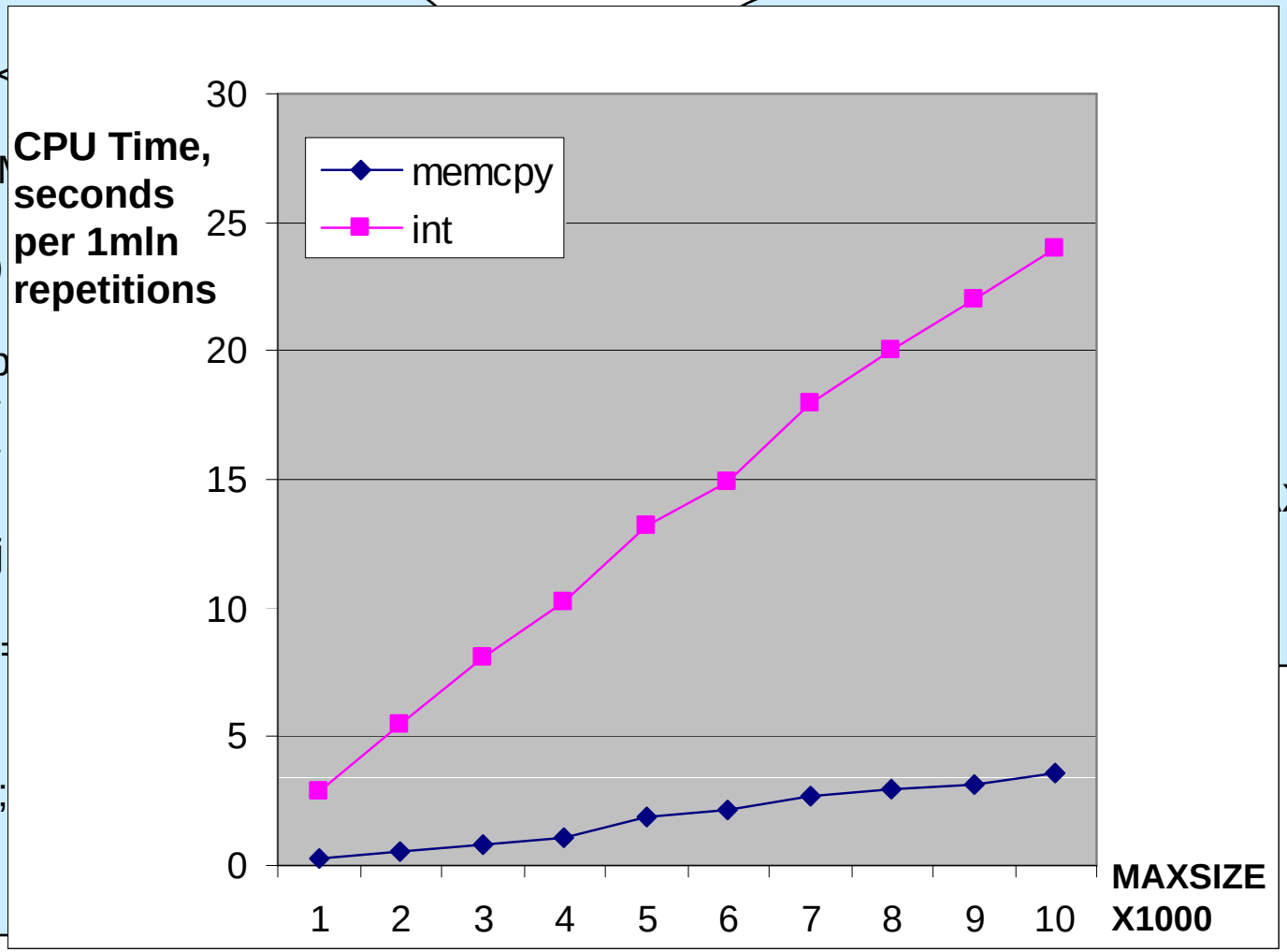
- **Assignment operator in L1A = (theData[i] >> 12) & 0xF; is at least one copy operation**

```
// C++ code
#include <memory.h>

const int MAXSIZE = 1000;

int main()
{
  int *a, *b, i;
  a = new int[MAXSIZE];
  b = new int[MAXSIZE];
  memcpy(b, a, sizeof(int) * MAXSIZE);
  return 0;
}

// Assembly code
// the address of a is in eax
// and the address of b in edx
mov ecx,  1000
mov esi, eax
mov edi, edx
rep movsd
```

Memcpy() timing benchmarks

```
#include <

const int M

int main()
{
  int *a, *b
  a = new
  b = new

for (long j
    {
      *a++ =
    }

  return 0;
}
```

XSIZE);



A word about raw data packing

- **Data stored in persistent objects (digis) can be converted into raw binary bite stream (Digi->Raw packing)**
- **Because each class stores data as bit fields all that is needed to create raw bit stream is *this* pointer and *size()* method that returns the size of the object:**

```
unsigned short * data() {return (unsigned short *) this;}
```

```
boost::dynamic_bitset<> rawDMBbits =
                    UshortToBitset(theDMBHeader.size(), theDMBHeader.data());
```

- **We use boost::dynamic_bitset<> class to handle bit streams**

Conclusions

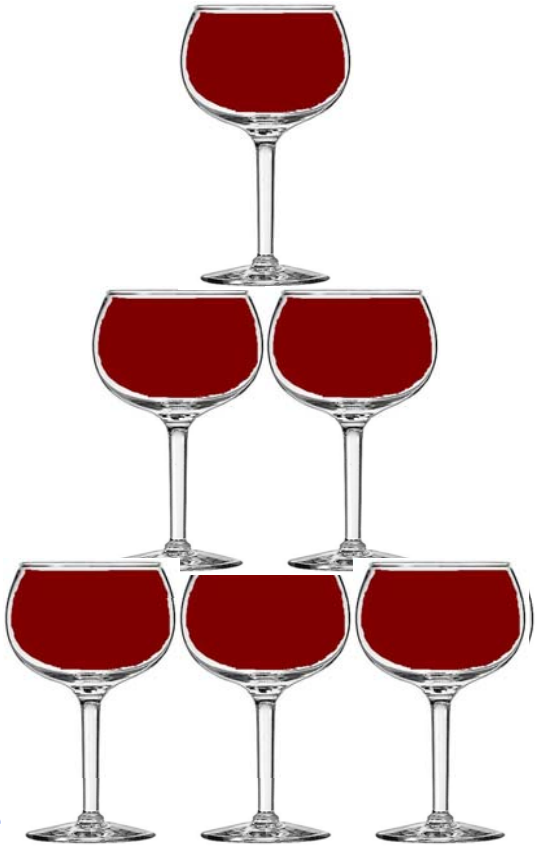
- **An order of magnitude in data unpacking speed can be gained by using bit fields instead of conventional bit shifts and masks**
- **The bit field data classes are usually easier to read and write – a huge advantage to help avoid probably the biggest source of unpacking problems – errors/typos/bugs in bit definitions**
- **Particular attention should be paid to questions related to memory and 32bit vs 64bit CPU architecture and endian-ness issues:**
  - ➔ **Recently CSC unpacking code was successfully moved from 32bit Scientific Linux 3 (SLC3) machines to 64bit SLC4**
  - ➔ **Extensive testing by Valgrind shows no memory problems**

Appendix

Pouring wine (data) slowly

Inefficient way: pour one at a time

Pouring wine (data) fast!



Efficient way: prearrange glasses and pour into all at once